Dependence Analysis of Functional Programs and its Applications

Submitted in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy
by

Prasanna Kumar K.
Roll No. 10405002

Advisors
Prof Amitabha Sanyal
and
Prof Amey Karkare

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY–BOMBAY
2018

Abstract

Static analysis of functional programs turns out to be more challenging than imperative programs. The main reasons being a compositional style of programming emphasizing creation of user-defined functions, use of algebraic data types and support for higher-order programming constructs. Techniques which work well for analyzing imperative programs do not suffice for functional programs. In this thesis, we formulate the problem of context-sensitive dependence analysis for first-order functional programs. However, we are interested in a more general notion of dependence called structure-transmitted dependence, which can answer questions such as: *For expressions e_1 and e_2 in a program, if σ_1 represents the parts of interest in the value of e_1, then which parts of the value of e_2 are required to compute σ_1*. We show that an analysis that is context-sensitive and models structure-transmitted dependences is undecidable. While a different formulation of this problem has already been proved to be undecidable, ours is an independent proof, both in terms of formulation and the reduction strategy employed.

Using the formulation we define an approximate dependence analysis which models structure-transmitted dependence precisely but over-approximates context-sensitivity. The resulting analysis is still precise enough to be useful for applications such as garbage collection and program slicing. In spite of the analysis being context-sensitive, we ensure that it is efficient—the body of a user defined function is analyzed only once, irrespective of the number of times the function is called. Given an expression e_1 and the parts of interest σ_1 expressed as a regular grammar, the result of the approximate analysis is a regular grammar corresponding to e_2 that answers the dependence question. We formally prove the soundness of our analysis. As applications, we use variants of the analysis for i) computing liveness of lazy first-order programs and use it for liveness-based garbage collection, and ii) static program slicing.

We first use our dependence analysis to compute liveness for lazy languages. A

variable is *live* if there is a possibility of its value being used in future computations and dead if it is definitely not used. A liveness-based garbage collector retains only references that are live as opposed to a reachability-based collector which retains all references that are reachable. Although it has been shown that liveness-based garbage collection is effective for eager first-order functional languages, extending the scheme to lazy languages is not straight forward. The reasons are: i) In a lazy language the point of evaluation of an expression can be determined statically, ii) references to values can escape their scope of declaration because of lazy constructors. Further, the garbage collector in a lazy language needs to handle unevaluated expressions (closures) during garbage collection. It has to make a liveness-based decision on which parts of the closure can be safely garbage collected. This is the first reported work that uses the results of an interprocedural liveness analysis to garbage collect both evaluated data and closures. We provide a proof of correctness of the liveness-based garbage collection scheme. Using a prototype that we have implemented, we show that the number of garbage collections and the peak memory required for executing the program reduces for all the programs in our test suite, and the total time spent doing garbage collection also reduces for many programs.

Program slicing refers to the class of techniques that delete parts of a given program while preserving certain desired behaviours. The desired behaviors are specified using what is called as the *slicing criterion*. Slicing can be used for debugging, software maintenance, optimization, program understanding and information flow control. We show how program slicing can be modelled as a dependence analysis problem. Applications such as program specialization, cohesion measurement and parallelization require the same program to be sliced with more than one slicing criterion. Using certain properties of our formulation of dependence analysis, we define a novel incremental method for slicing programs, i.e., slicing the same program with different input slicing criteria. We show the performance benefits of incremental slicing by implementing a slicer capable of slicing incrementally and non-incrementally. In the interest of completeness, we handle higher-order programs by converting them into first-order programs through a process called firstification. We run dependence analysis on the first-order program and obtain its slice. The resulting sliced program is mapped back to the original higher-order program. As an example implementation, we extend our slicing algorithm to handle higher-order programs.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Static analysis is a collection of techniques that finds useful information about programs. Such information has a variety of uses—debugging, optimization and program verification are examples. A static analysis consists of two parts: i) a suitable representation for the information being computed by the analysis, which we shall generally call *dataflow information*, and ii) a fixed point iteration over a representation of the program to compute the dataflow information. Given the differences in the imperative and the functional programming styles, the kind of information sought and consequently the nature of the analysis differ, in general, for programs written in the two styles.

1.1 Analysis of imperative programs

A common example of an analysis in imperative languages is *reaching definitions* [7, 8]. This determines the definitions of a variable that may reach a particular program point. The dataflow information in this case may be represented as sequence of boolean values, also called a bit-vector, each value representing a definition of a variable at a program point. In contrast, analyses which compute properties of programs that manipulate the heap through recursive types require a more complex dataflow representation. This is because such programs could be used to access unbounded regions of the heap memory and the dataflow information is usually an abstraction of regions of the heap, often in the form of graphs. Alias analysis or pointer analysis[9, 24, 28, 42, 48, 49, 52, 53, 95], liveness analysis[5, 44, 46, 47] and shape analysis[84, 85] are examples of such analyses.

The fixed point iteration to compute the analysis results can be performed on a

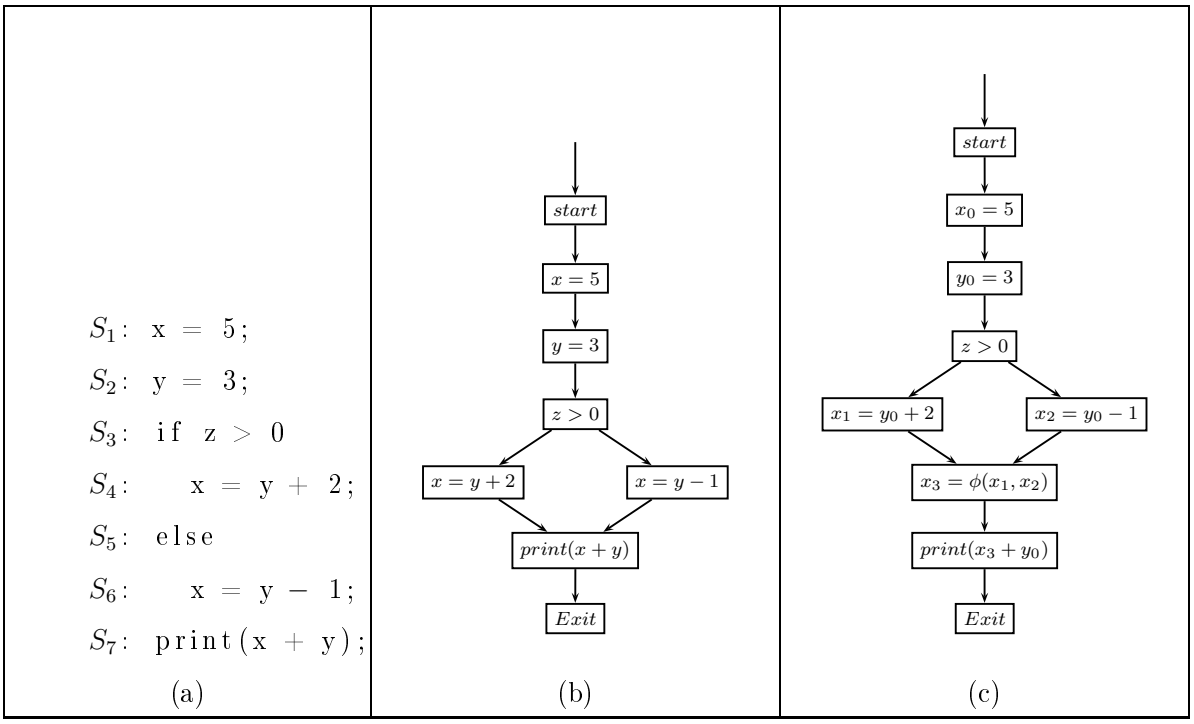


Figure 1.1: Program representations for static analysis. (a) An example program (b) Control Flow Graph of the program in (a) (b) SSA representation of the program in (a)

structure that models the program being analyzed. The program is usually represented as a graph with the statements in the program as nodes and relations such as data/control flow among statements as edges. Examples of such representations are the *Control Flow Graph (CnFG)*, *Static Single Assignment (SSA) form* and *Program Dependence Graph (PDG)* [27]. Figure 1.1 shows an example program and its representation as a CnFG is shown in Figure 1.1(b). In this graph, nodes represent statements in the program. and edges represent possible flow of control from one statement to another.

However, notice that the CnFG may separate statements that are computationally close to each other. As an example, statement S_2 defines a variable y that is used in S_4. The pair (S_2, S_4), called a *def-use* pair, represents a direct *data dependence* between S_2 and S_4. In general, data dependence is a transitive relation. In the example program, S_7 is data dependent on S_6 due to the use of x. S_6 is data dependent on S_2 due to its use of y and hence transitively, S_7 is data dependent on S_2. Since the definition and use might be separated by other statements, data dependences are not obvious in a CnFG.

There is yet another sense in which a statement S_i can depend on S_j. The value computed by S_j decides whether the statement S_i gets executed or not. In such a case we

say that S_i is control dependent on S_j. In the example program, execution of statement S_4 and S_6 depends on what S_3 evaluates to. Therefore both S_4 and S_6 are control dependent on S_3.

There are two program representations that attempt to capture dependences directly namely *Program Dependence Graph* and *Single Static Assignment (SSA)*. In the SSA form, each variable being defined is renamed to a unique name, its uses are connected to the use by being renamed to the same name. However, a PDG captures both data dependence as well as control dependence in the program through direct edges between constituent nodes. Applications like slicing of programs requires information about both data and control dependences. For such applications we use a PDG. PDGs have wide applicability in imperative languages [27, 38]. In this thesis we explore the concept and utility of dependence in the context of functional languages.

1.2 Analysis of functional programs

An example of an analysis for functional programs which does not have a counterpart in the imperative world is *strictness analysis*. This analysis is applicable in lazy functional languages in which expressions are not evaluated unless their values are required. Therefore, arguments to functions are passed as unevaluated expressions (thunks or closures). This is a source of space inefficiency as closures may require more space to store than values. This may also affect the execution time since the garbage collector may have to be invoked more often. One way of improving the efficiency is to identify arguments which are guaranteed to be evaluated inside a function and then evaluate them before a call to the function. Thus, strictness information is associated with the arguments of a function definition and comes in two forms. If the argument is a scalar, the analysis says whether the argument is guaranteed to be evaluated or not. And if the argument is a structured data such as a list, the analysis also indicates the extent of guaranteed evaluation—no evaluation, head only, spine only and full evaluation.

For now, we assume the reader's familiarity with the Scheme programming language. Consider the program in Figure 1.2, the **main** expression creates a list `x` which contains two expressions e_1 and e_2. From the program, it is clear that the result of the main expression is the **car** part of the result returned by function call (**map square** `x`). Hence,

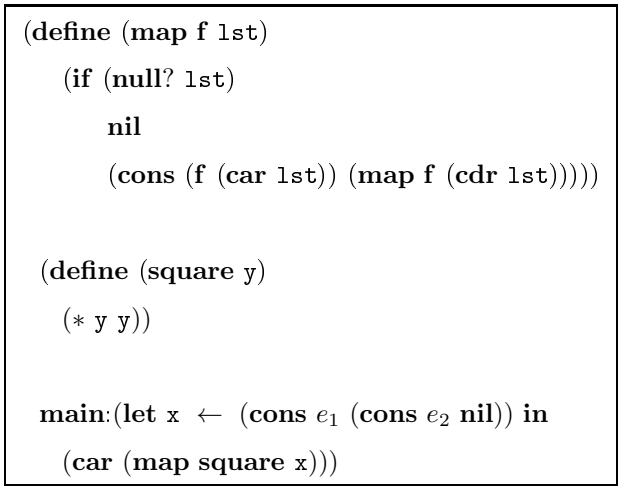


Figure 1.2: Example for strictness analysis

this information will be propagated through the body of function **map** and it can be determined that the expression (**f** (**car** `lst`)) will definitely be evaluated. Since the actual argument being passed to the function **map** is `x`, we get the information that the **car** part of `x` which is e_1 will definitely be evaluated. Therefore, we can eagerly evaluate e_1 safely without violating the lazy semantics of this language.

Judged by this example, analysis of functional programs could differ from their imperative counterparts in the following ways.

1. *In the nature of the information sought:* In the world of imperative languages, it is uncommon to seek information like: *Is the argument of a function likely to be evaluated along all paths in the function, and if so, what is the common extent of evaluation along all paths?*.
2. *Analysis for functional programs must necessarily be interprocedural:* Limiting the analysis to intraprocedural levels with conservative approximation at procedural boundaries may not yield significant benefits.
3. *Identifying structure-transmitted data dependences [78] is important:* Consider a variable `z` bound to (**cdr** (**cons** `x` `y`)). The fact that the value of `z` does not depend on `x` requires the analysis to incorporate the identity (**cdr** (**cons** `x` `y`)) = `y`. While this is important for the precision of the analysis, identifying the constructor-selector interaction interprocedurally is undecidable.
4. *Additionally, control flow is hard to figure out for functional programs:* This is due to the presence of higher-order functions and in the case of lazy languages, closures.

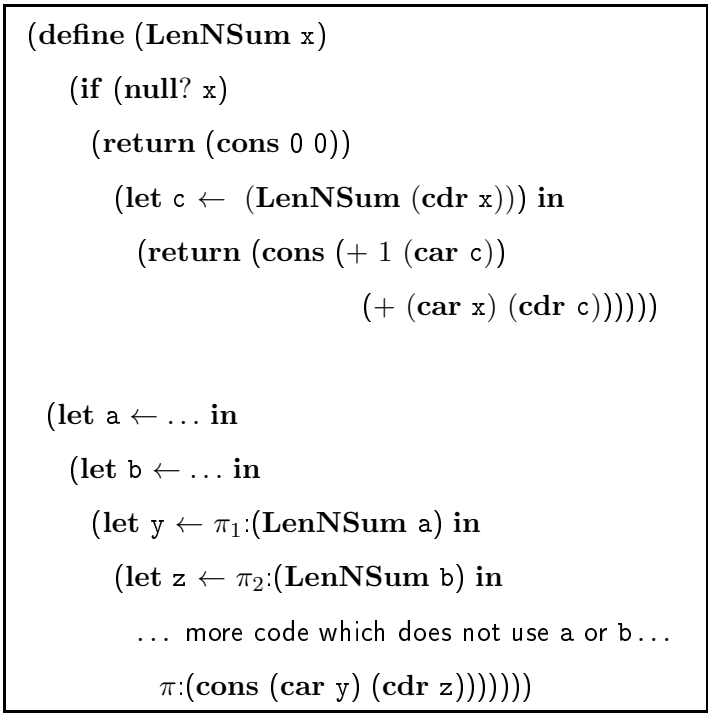


Figure 1.3: Motivating example for dependence analysis of functional programs

1.3 Dependence analysis of functional programs

An analysis that is common to both imperative and functional languages is "dependence analysis". In the imperative domain, for each variable v in the program, dependence analysis computes the set of variables on which v is data or control dependent. Identifying dependences (data or control) in a program is key in many program optimizations and applications such as slicing. Knowing the dependences among variables, it is possible to optimize the program by improving its run time or memory usage. For example, if we can find out that a certain value is computed but never used, we can safely remove the code corresponding to the generation of this value. This results in faster execution as the code for generating the value is not executed. Also, the modified program uses less memory since the memory required for storing the value can be avoided. In the functional domain, we need to compute dependences among expressions instead of variables. Using the example in Figure 1.3, we describe some applications of dependence information in functional programs.

Consider the example in Figure 1.3, the function **LenNSum** written in a Scheme like language, takes a list of integers as input and computes both the length of the list as well as the sum of the elements of the list. The symbol π is not part of the program

and is used to denote a program point. For an empty list it returns (**cons** 0 0), signifying that the length of an empty list as well as the sum of the elements of an empty list are both 0. For any non-empty list, the function recursively computes the length and sum of the **cdr** part of the input list and increments the **car** part of the result by 1 to compute the length and adds the **car** element of the input list to compute the sum for the input list. The function creates a **cons** cell to enclose the computed length and sum values and returns it. Notice, the length of the list does not depend on the values of elements of the list.

In an eager language, there are no uses of `a` beyond π_1 and of `b` beyond π_2. Therefore, we can safely garbage collect the memory associated with list `a` after π_1 and `b` after π_2. A garbage collector which collects only references which become unreachable would not have been able to collect `a` and `b` as they would still be in scope and hence reachable. However, the situation gets interesting in the case of lazy language. In a lazy language, a let expression does not trigger the evaluation of the expression bound to the let variable as soon as it is encountered. Instead, it creates a closure and defers its evaluation until its value is actually required. Therefore, in the example program, assuming that the output of **main** is being printed, only the length of the list `a` is required. Therefore, unlike in an eager language, the function **LenNSum** does not evaluate the expressions which compute the sum of the list `a`. Thus, the final output does not depend on any element of the list `a`. Given this information, a garbage collector could efficiently collect the memory corresponding to the elements of `a` any time after the creation of the list `a`.

Another application of dependence information is program specialization. The function **LenNSum** computes both the sum and length of the input list. If we need a specialized function that just computes the length of the list, we can remove expressions that definitely do not contribute to the **car** part of its output from the body of **LenNSum** to get the specialized function.

In this thesis, we consider the problem of static analysis of functional programs to compute dependences. The analysis is interprocedural and is defined for a Scheme-like first-order functional language. Lists are the only user-defined data structures that are supported as other data-structures can be modelled using lists. Extending our analysis to other user-defined data structures does not present any conceptual difficulties. Our analysis strikes a balance between precision and efficiency by computing function sum-

maries for user-defined functions and using them to compute dependences for function call expressions.

1.4 Contributions of this thesis

This thesis presents an efficient and reasonably precise interprocedural dependence analysis for functional programs. While Reps [78] has shown that an interprocedural dependence analysis that is context-sensitive and precisely models structure-transmitted data dependences is undecidable, we provide an independent proof of undecidability and use our formulation to propose an approximate dependence analysis. The analysis proposed in the thesis precisely models structure-transmitted data dependences while relaxing the context-sensitivity requirement in some cases to allow the analysis to become decidable. Thus, our analysis can be both context-sensitive and precisely model structure-transmitted data dependences in most cases. The loss of precision when the context-sensitivity requirement is relaxed is still tolerable to be useful in applications like liveness-based garbage collection and slicing. To compute dependences, we generalize the analysis described by Asati et al [12] that computes liveness in eager first-order functional programs. The dependence analysis is defined over a language which has lazy semantics. This enhances the applicability of the analysis by allowing us to perform liveness analysis of lazy languages and perform static program slicing.

1.4.1 Dependence analysis

In the context of imperative languages, dependence analysis answers the question: *Given a statement, say S_1, what are the statements S_2 that it depends on?* The definition can be generalized and made more interesting in the case of functional languages, especially when the value computed by the program is an algebraic datatype. Let e be an expression which evaluates to an algebraic datatype and σ denote a substructure of this value. The notion of dependence that we wish to address is: *Given a part (or substructure) σ of the value an expression e, what parts σ_i of the value of other expressions e_i decide the value of σ part of the value of e?* As a concrete example, for the program in Figure 1.4, our analysis should yield the information that the only part of list `c` in the let expression π_{12}

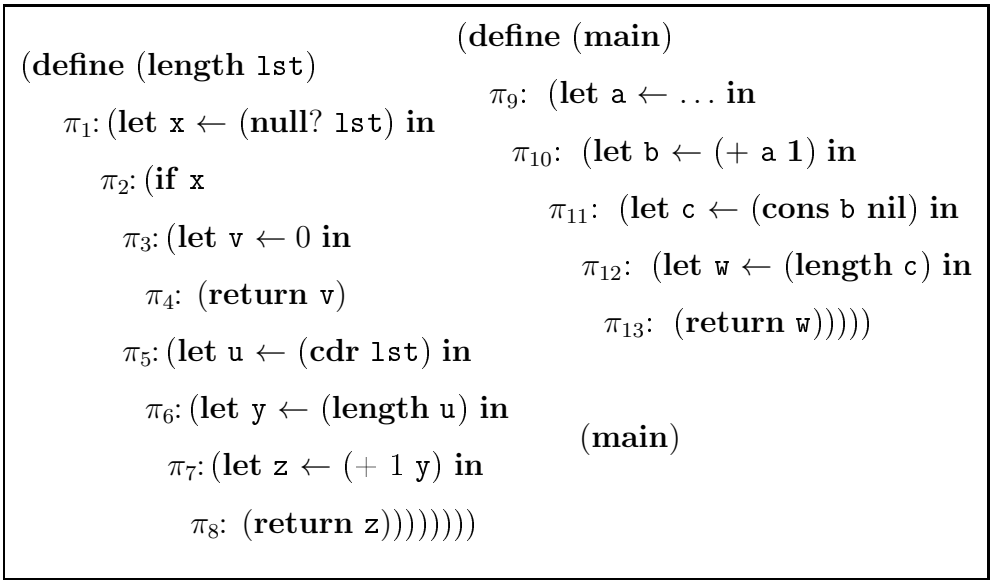


Figure 1.4: An example program

that decides the value of (**main**) is its spine[1], the elements of the list are not required.

We call substructures of a value that are of interest as *demands* and represent them as follows. The demand $\emptyset$ indicates that no part of the value is of interest. For an integer, we indicate that the value is of interest by using the demand ϵ. In the case of algebraic datatypes like lists, we can construct a tree and label its branches by selectors, using the notations **0** and **1** to represent selections using **car** and **cdr** respectively. This is shown in Figure 1.5. The substructure of interest can then be identified by a path from the root of the tree. As an example, the substructure represented by the highlighted path in blue, is represented by the set $\{\mathbf{10}\}$. Similarly, the spine of the list (highlighted in red) is represented by the set $\{\mathbf{11}\}$. If the size of the list is unknown, then the spine can be approximated by the infinite set $\{\epsilon, \mathbf{1}, \mathbf{11}, \ldots\}$ or, $\mathbf{1}^*$ in short. Paraphrasing our earlier observation, a demand of ϵ on (**main**) is decided by (or depends on) the demand $\mathbf{1}^*$ on `c`, and, interestingly, $\emptyset$ (or no part) of the value of (+ `a` 1).

The idea behind dependence analysis is to propagate demands from an outer expression to inner expressions, in the example from the body of **main** to the expressions `c` and (+ `a` 1). We give rules to do this outside-in propagation for the **let** and the **if** expressions, and the built-in operators, **cons**, **car**, **cdr**, **null**? and +. However, we also uniformly extend the outside-in propagation principle to user defined function calls $(f\ x)$,

[1]The *spine* of a list is the substructure obtained by starting with at the root and a performing a series of **cdr** selections reaching the end of the list. For the list in Figure 1.5, the edges in red is the spine.

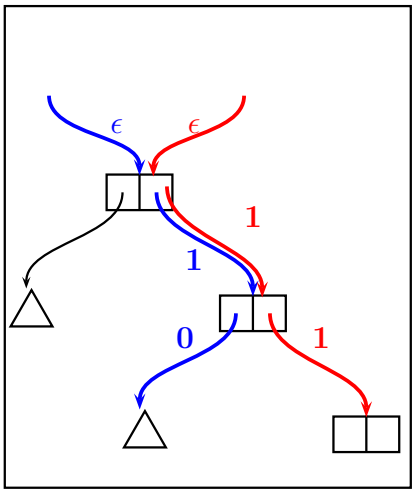


Figure 1.5: A list represented as a tree with edges labelled with the corresponding selector operations.

by computing a transfer function, denoted $\mathbb{DS}_f$. If σ is the demand on a function call $(f\ x)$, then $\mathbb{DS}_f(\sigma)$ is the propagation of this demand to its argument x. Our proposed analysis gives $\mathbb{DS}_{\mathbf{length}}$ for a non-null σ as:

$$\mathbb{DS}_{\mathbf{length}}(\sigma) = \epsilon \cup \mathbf{1}\mathbb{DS}_{\mathbf{length}}(\epsilon)$$

The important point to note that the unknown in this equation is the function $\mathbb{DS}_{\mathbf{length}}$, and the reader can verify that $\mathbb{DS}_{\mathbf{length}}(\sigma) = \mathbf{1}^*$ is a solution of this equation. What this means is that any non-null demand on (**length** `c`) will propagate a spine demand on `c`. This matches our intuition as the **length** function recursively traverses the spine of its argument till the end of the list. Therefore, the demand on `c` is $\mathbf{1}^*$, and, since `c` is bound to (**cons** `b` **nil**), the same demand is transferred to this expression. Going inwards still further, since `b` is the head of the list (**cons** `b` **nil**) with the demand $\mathbf{1}^*$, the demand on `b` is $\emptyset$, a fact that we infer through algebraic rules. This $\emptyset$ demand is transferred to (+ `a` 1), and we conclude that the output of **main** does not depend on any part of (+ `a` 1).

More specifically, our contributions in this part are as follows:

1. We generalize liveness analysis to a more general notion of dependence, and propose a context-sensitive interprocedural analysis that precisely models structure-transmitted data dependences to compute dependences in a program.
2. Independent of Reps [78] and using a different reduction, we show that context-sensitive

and precise modelling of structure-transmitted data dependences is undecidable. Using our formulation we propose an approximate analysis which precisely models structure-transmitted data dependences but relaxes context-sensitivity requirement in some cases.

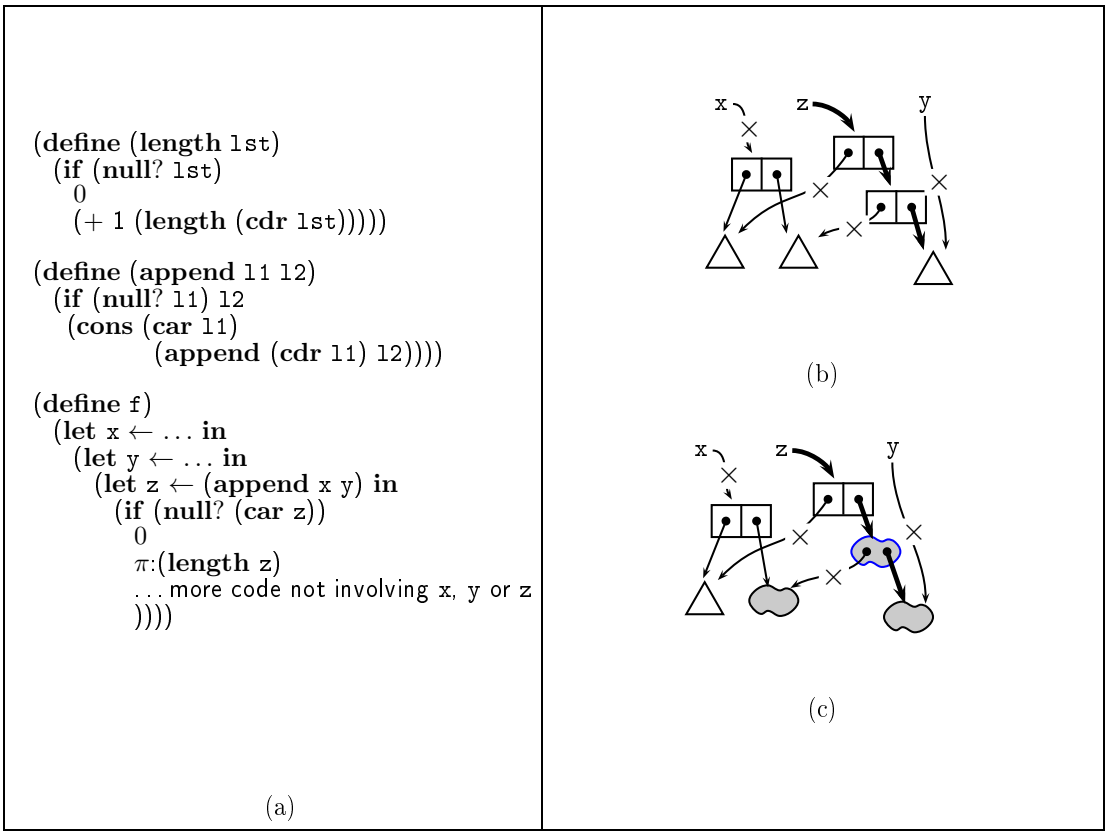
3. The analysis results in recursive equations, where the unknowns are transfer functions such as $\mathbb{DS}_f(\sigma)$. The solution takes the form $\mathsf{DS}_f\,\sigma$ (DS_f concatenated with σ, with concatenation lifted naturally to sets of strings). Here DS_f is the start symbol of a CFG with two fixed non-CFG productions[2] We prove that the membership problem of the resulting grammar is undecidable. We get around the problem by approximating the CFG by a regular grammar.
4. Based on a demand driven operational semantics for the language, we prove the soundness of the analysis.

1.4.2 Liveness-based garbage collection for lazy languages

The runtime system of most functional languages includes a garbage collector to reclaim memory, however empirical studies on Scheme [45] and Haskell [82] programs have shown that garbage collectors leave uncollected a large number of memory objects that are reachable but are not live, i.e. these memory objects are guaranteed not be used when execution resumes from the current state after garbage collection. This results in unnecessary retention of memory which can be safely garbage collected. The situation is even worse in the case of lazy functional languages as they might have to carry large closures (runtime representations of unevaluated expressions) instead of values. To remedy this, Asati [12] proposes a liveness-based garbage collector instead of a reachability-based collector to increase the number of cells garbage collected. However, the proposal was for a eager language.

We use the example in Figure 1.6 to demonstrate the benefits of a liveness-based garbage collection scheme and also the challenges faced in implementing a liveness-based collector for lazy languages. We represent the heap as a graph in which a node either represents atomic values (**nil**, integers, etc.), or a **cons** cell containing **car** and **cdr** fields,

[2]We reiterate the differences between $\mathbb{DS}_f$ and DS_f. $\mathbb{DS}_f$ is a transfer function and is the unknown in the equation generated by the analysis. DS_f is a grammar symbol representing a set of strings, and is a part of the solution. The solution of $\mathbb{DS}_f$ maps a demand σ to $\mathsf{DS}_f\,\sigma$.



denotes a closure. Thick edges denote live links. Traversal stops at edges marked × during garbage collection for a liveness-based collector.

Figure 1.6: An example program and its memory graph. (b) represents the heap state in an eager language and (c) represents the heap state in a lazy language.

or a closure (represented by shaded clouds) in the case of lazy languages. Edges in the graph are *references* and represent values of variables or fields. The situation in the case of eager languages is shown in Figure 1.6(b). At program point π, the liveness associated with `z` is $\mathbf{1}^*$ and with `x` and `y` it is $\emptyset$ as there are no more uses for `x` and `y` beyond π. Thus, if a GC takes place at π with the heap shown in Figure 1.6(b), a liveness-based collector will preserve only the cell referenced by the spine of `z`.

In contrast, Figure 1.6(c) shows the lists `x` and `z` partially evaluated due to the **if** condition (**null**? (**car** `z`)). Due to this evaluation, the **car** field of `z` points to the **car** of `x` and the **cdr** field of `z` points to the closure (**cons** (**car** `x`) (**append** (**cdr** `x`) `y`)) (shown as the bubble outlined in blue). Here we face a situation that is different from eager evaluation in the following senses: (i) Laziness dictates that (**length** `z`) will be evaluated on demand, so it is statically not possible to figure out where this evaluation will take

place. In fact it may even take place beyond the scope in which z has been declared, indeed even outside the function f. (ii) Given that the spine of z has not been evaluated yet, how would a liveness-based garbage collector, if invoked at π, collect the spine of z?

Our solution to these problems is as follows. We think of the free variables inside a closure as root set variables, and carry their liveness information inside the closure. Thus the closure (**length** z) carries the liveness of z as $\mathbf{1}^*$. Second, if a variable is not fully evaluated during garbage collection, we have devised a mechanism of garbage collecting (parts of the) closure. Roughly the idea is as follows: If the demand on, say (**length** z) is $\emptyset$, the closure is garbage collected. Otherwise, we consult the recorded liveness to garbage collect whatever is bound to z. If z happens to be evaluated, we use the recorded liveness $\mathbf{1}^*$ to garbage collect the value, else, if it is bound to a closure, we repeat the same process as we did for (**length** z).

We use a variation of dependence analysis to compute liveness information and store them as Deterministic Finite Automata (DFA) at program points of interest. Since our application is garbage collection, the program points of interest are the ones which could potentially trigger a garbage collection, which, for lazy languages, is the point where a **let** expression requests memory to create a closure that is bound to the **let**variable. If the garbage collector is invoked at any of these program points, it uses the associated automata to curtail reachability during marking phase. This results in an increase in the garbage reclaimed and consequently in fewer collections.

Our contributions in this part are:

1. Whereas the idea of using static analysis to improve memory utilization has been shown to be effective for *eager* languages [12, 36, 41, 56], a straightforward extension of the technique is not possible for lazy languages, where heap-allocated objects may include closures. We define a liveness analysis for first-order lazy languages. To make liveness-based GC effective for lazy languages we extend it to closures apart from evaluated data. Closures carry liveness information of its free variables which is used by the garbage collector during garbage collection. As an optimization, to keep the closure liveness precise, we update it during the execution.
2. To prove the soundness of our method, we modify the demand-guided semantics introduced for dependence analysis by introducing an updatable Heap as part of the state. We also simulate a Garbage Collection in the semantics which goes to a special state

called BANG if a reference which has been declared dead by our analysis. The soundness analysis consists of proving that for any program, the execution of the demand-guided semantics cannot enter a BANG state.

3. Using a prototype implementation we demonstrate the benefits of liveness-based garbage collection for lazy languages, increase in the garbage reclaimed and consequently in fewer collections. Because a liveness-based collector identified more cells that would not be used, the peak memory use also improved for all the programs. Our experiments show up to 10X reduction in the number of garbage collections and 20X reduction in peak memory requirements.

1.4.3 Program slicing

Program slicing is a powerful technique with applications ranging from debugging, software maintenance, optimization, program analysis and information flow control. Program slicing refers to the class of techniques that delete parts of a given program while preserving certain desired behaviours. The desired behaviors are specified using what is called as the 'slicing criterion'. We consider one such version of slicing where the slicing criterion identifies parts of the final output of the program that the user of the slicing tool may be interested in, and the goal is to produce the parts of the program which affect only the parts of the output identified by the slicing criterion. Program specialization, parallelization, dead code analysis and cohesion measurement are examples of such applications.

In general, a slicing criterion is modeled as a pair $\langle e, \sigma \rangle$, where e represents an expression in the program P and σ represents the parts of the value of e that is of interest. The goal of slicing is to identify the set of expressions belonging to P which may affect the parts identified by σ. The slicing problem thus can be formulated as an instance of *dependence analysis* as the question: Given a slicing criterion $\langle e, \sigma \rangle$, what are the expressions e_i in P such that σ of e depends on σ_i of e_i, and σ_i is not $\emptyset$? While the general form of dependence analysis can answer the question for any expression e, we consider the special but useful case in which e is the main expression (**main**). Figure 1.7a shows a simple program in a Scheme-like language taken from [79]. It takes a string as input and returns a pair consisting of the number of characters and lines in the string. Figure 1.7b shows the program when it is sliced with respect to the first component of the output pair, namely the number of lines in the string (`lc`). All references to the count

```
(define (lcc str lc cc)
    (if (null? str)
        (return (cons lc cc))
        (if (eq? (car str) nl)
            (return (lcc (cdr str) (+ lc 1) (+ cc 1)))
            (return (lcc (cdr str) π1:lc π2:(+ cc 1))))))
(define (main)
    (return (lcc ... 0 0))))
(main)
```

(a) Program to compute the number of lines and characters in a string.

```
(define (lcc str lc □)
    (if (null? str)
        (return (cons lc □))
        (if (eq? (car str) nl)
            (return (lcc (cdr str) (+ lc 1) □))
            (return (lcc (cdr str) π1:lc π2:□)))))
(define (main)
    (return (lcc ... 0 □))))
(main)
```

(b) Slice of program in (a) to compute the number of lines only

```
(define (lcc str □ cc)
    (if (null? str)
        (return (cons □ cc))
        (if (eq? (car str) nl)
            (return (lcc (cdr str) □ (+ cc 1)))
            (return (lcc (cdr str) π1:□ π2:(+ cc 1))))))
(define (main)
    (return (lcc ... □ 0))))
(main)
```

(c) Slice of program in (a) to compute the number of characters only.

Figure 1.7: A program in Scheme-like language and its slices. The parts that are sliced away are denoted by □.

of characters (`cc`) and the expressions responsible for computing `cc` only have been sliced away (denoted $\square$).

To produce a slice for a given program, we use the slicing criterion (a set of strings over $(\mathbf{0}+\mathbf{1})^*$) as a demand on the main expression (**main**), and compute the demands on each expression in the program. Any expression which gets a $\emptyset$ demand can be sliced from the program. Since we model the slicing problem as an instance of dependence analysis, we inherit both its advantages and its weaknesses. One of the weaknesses is the large time required for automata construction. This was acceptable in the case of garbage collection as the automata were created only once and the same would be consulted whenever a garbage collection was triggered. However, in case of slicing the automata have to be reconstructed every time the slicing criterion changes (even when the program remains same). Program specialization is an example application where the same program is sliced with different criteria. We thus turned to the problem of incremental slicing that is useful in such situations.

Incremental slicing

The program in Figure 1.7a can also be sliced with respect to the second component of the output (the character count). The resulting program in which all expressions related to computation of `lc` have been removed is shown in Figure 1.7c. If we use the non-incremental slicing method, the whole procedure of dependence analysis has to be repeated for the new slicing criterion. Applications such as program specialization, cohesion measurement and parallelization which require the same program to be sliced with more than one slicing criterion can benefit from a slicing method which avoids repeated computation of information. We propose an incremental algorithm based on dependence analysis that avoids this repetition of computation when the same program is sliced with different criteria.

The incremental slicing algorithm involves a one-time precomputation step that computes information which is common to all slicing criteria. The key idea is that for each expression, the precomputation step computes the set of *all slicing criteria* which keeps the expression in the slice. It is an interesting fact that this can be done by considering only one specific criterion, namely $\{\epsilon\}$. The resulting set can be represented as a finite state automaton. Now, given any other criterion (also represented as a finite state

automaton), the incremental slicing procedure simply finds the intersection of the two automata. If the language of the resulting automaton is $\emptyset$, the expression can be sliced out. Notice, that even when the slicing criterion changes the automata computed in the precomputation step do not change and hence they don't need to be recomputed.

Finally, for the sake of completeness we describe a method to extend our dependence analysis to handle higher-order programs. We first convert the input higher-order program to its equivalent first-order program by a process called firstification [60] while maintaining a mapping between the original program and the firstified program. We perform dependence analysis on the firstified program and obtain the results. Using the mapping generated during the firstification process, we transfer the demands generated on the firstified program to the original program. We extend the static slicer to handle higher-order programs.

Our contributions in this part are:

1. We use dependence analysis to statically slice functional programs. We define a novel incremental slicing mechanism which is very efficient when the same program is sliced multiple times. The correctness of our static slicing algorithm follows from the correctness of dependence analysis.
2. We formally prove that our incremental slicing is sound with respect to non-incremental slicing.
3. We extend slicing to handle higher-order programs by performing firstification.
4. We have implemented a prototype of a slicer that can run in incremental or non-incremental mode. The slicer can handle both first-order and higher-order programs written in a Scheme like language. Our experiments confirm that the incremental computation runs orders of magnitude faster than the non-incremental version. We obtain nearly 1000X speed in nearly all of the experiments.

1.5 Related work

The dependence analysis that we defined involved starting with a demand σ on an expression e and computing demands on each sub-expression of e. Similar approaches of taking an abstract value and propagating its effects "inwards" have been used in different analysis. Wadler [101] uses projection functions which he calls "contexts" to perform strictness

analysis. Given the context on the result of a function f, strictness analysis computes the parts of the arguments of f that could be safely evaluated eagerly. Wadler [100] shows how contexts can be used to compute the time complexity of lazy programs. The propagation of abstract values can also be done in the "outwards" direction, starting from arguments of an expression to the result of the expression. Binding time analysis [40, 61, 65] is an example of such an analysis. It is used in partial evaluators to determine the parts of the program can be evaluated if some input is known. In the rest of this section we mainly discuss work which has similar applications to our analysis, improving garbage collection using static analysis and static slicing of functional programs.

There have been different approaches to improving memory utilization in functional programs. An interesting approach by Mohnen [62] uses abstract interpretation to handle garbage collection of nested lists. A list having n levels is abstracted to an n-tuple, a boolean denoting the possibility of sharing between any element at each level in the list and the result of the function to which the list is passed as a parameter. A false value in the tuple indicates that values at that level are not shared with the return value and hence can be garbage collected. This leads to very coarse approximations as the use of a single cell will make the whole list at that level live. The approach due to Lee et. al. [55, 56] uses *memory types* to describe usage of heap cells and achieves context sensitivity by using dynamic flags passed as extra arguments to functions to collect cells inside function bodies. Passing different values from different call sites for the dynamic flags allows the same function to have different deallocation behaviors. A practical approach involves copying only the heap objects whose root variables are live [6]. The drawback of this approach is that an entire object reachable from a live root variable is considered live, even if some parts of it are never used. For example, even when only the spine of a list is live (used as an argument to the **length** function) all its elements will also be copied. Asati et al [12] describe a liveness-based garbage collector for a first-order eager functional language based on an analysis similar to ours. It demonstrated the utility of a liveness-based collector over a reachability-based collector for a language which had support for heap based lists. The analysis described was limited to liveness analysis of first-order data values in an eager language. In this thesis, we generalize their analysis and demonstrate its utility by implementing a liveness-based garbage collector for a lazy language and then using the same analysis to statically slice functional programs.

Most of the efforts in slicing have been for imperative programs. The surveys [16, 92, 97] give good overviews of the variants of the slicing problem and their solution techniques. A natural way is to use data and control dependences between statements to compute the slice. The program to be sliced is transformed into a graph called the program dependence graph (PDG) [39, 72], in which nodes represent individual statements and edges represent dependences between them. The slice consists of the nodes in the PDG that are reachable through a backward traversal starting from the node representing the slicing criterion. Horwitz, Reps and Binkley [39] extend PDGs by defining a System Dependence Graph (SDG) to handle interprocedural slicing.

Silva, Tamarit and Tomás [93] adapt SDGs for functional languages, in particular Erlang. The adaptation is straightforward except that they handle dependences that arise out of pattern matching. Because of the use of SDGs, they can manage calling contexts precisely. However, as pointed out by the authors themselves, they fail to handle constructor-selector interactions. The slicing technique that is closest to ours is due to Reps and Turnidge [79] which uses projection functions to specify a slicing criteria. The slicing criterion is propagated backwards to all subexpressions of the program. After propagation, any expression with the projection function $\perp$ (corresponding to our $\emptyset$ demand), is sliced out of the program. Liu and Stoller [57] use a similar method for dead code analysis and elimination. Both these methods, unlike ours, do not derive context-independent summaries of functions. Thus, we do not see any way of computing multiple slices incrementally.

A graph based approach has also been used by Rodrigues and Barbosa [80] for component identification in Haskell programs which is very coarse for our purpose. Rodrigues and Barbosa [81] use program calculation in the Bird-Meerteens formalism for obtaining a slice. Given a program P and a projection function ψ, they calculate a program which is equivalent to $\psi \circ P$.

1.6 Organization of the thesis

We formalize the notion of an interprocedural context-sensitive dependence analysis that precisely models structure-transmitted data dependences in Chapter 2. We prove that such an analysis is undecidable using a non-standard operation semantics called demand-

guided semantics. In Chapter 3, we use the dependence analysis formulation defined in the previous chapter to devise an approximate dependence analysis which models structure-transmitted data dependences precisely but relaxes context-sensitivity for some cases. We prove its soundness with respect to demand-guided semantics. The first application of our dependence analysis, a liveness-based garbage collection scheme for a first-order lazy language is described in Chapter 4 and proved correct. In Chapter 5, we show how functional programs can be statically sliced using dependence analysis. We also present an efficient slicing technique called incremental slicing for slicing the same program with different criteria. We prove its correctness with respect to the non-incremental slicing algorithm. We describe a way to extend our analysis to handle higher-order programs. We demonstrate it by extending our static slicer to handle higher-order programs. Finally, we summarize our contributions and discuss potential extensions to our work in Chapter 6.

Chapter 2

Dependence analysis of functional programs

In this chapter, we first look at the notion of dependence for imperative programs. We then introduce an useful generalization of the notion of dependence for functional programs by introducing a concept called *demand*. A demand describes a part of the value of an expression, whose dependence is of interest and the analysis computes part of values of other expressions that this expression depends on. We then describe the syntax of a first-order functional language without imperative features that we shall use throughout the thesis. We then formally specify the dependence analysis problem—an algorithmic solution of the problem would compute the generalized notion of dependence mentioned above for programs written in this language. We show that the dependence analysis problem is undecidable and therefore there is no such algorithm. In the next chapter, we propose an approximate algorithm to compute dependence and prove its soundness.

2.1 Dependence Analysis of Imperative Programs

In the context of imperative languages, dependence analysis answers the question: *Given a statement, say S_1, what are the statements S_2 that it depends on?* Dependences between statements arise because of two distinct reasons. In the example in Figure 2.1, the statement S_7 assigns the value `x+y` to `z`. Thus any change in the values of `x` or `y` will affect the value of `x+y` and therefore the value assigned to `z`. Since statements S_3, S_5 and S_6 define `x` and `y`, S_7 is said to be dependent on S_3, S_5 and S_6. This kind of dependence

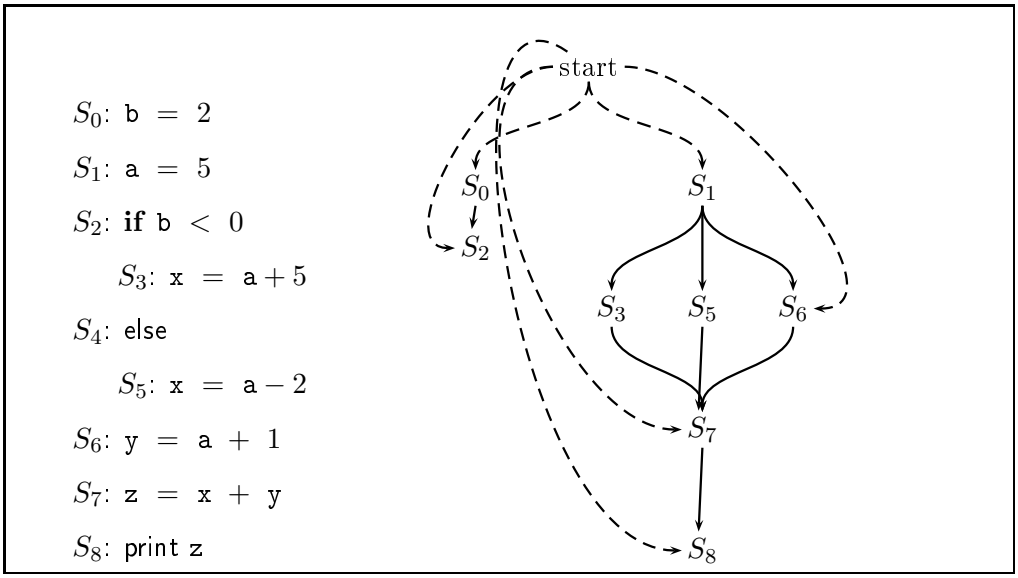


Figure 2.1: Imperative program and its PDG. Solid lines indicate data dependence and dashed lines control dependence.

is called *data dependence*. In summary, if S_i defines a value that is directly or transitively used by S_j we say that S_j is data dependent on S_i.

There is a second reason due to which a statement can be seen as depending on another. For the present discussion, we shall regard a boolean expression representing the condition of a *if* as a statement. In the example, the selection of S_3 or S_5 for execution depends upon the condition `b > 0` in S_2. We say that S_3 and S_5 are *control dependent* on S_2. This is viewed as a dependence because the value of `x` that reaches S_7 depends on the statement selected by S_2. In general, we say that S_j is control dependent on S_i if S_i decides whether S_j is executed or not.

Dependence is a transitive relation. In the example, S_7 is data dependent on S_3, and S_3, in turn, is data dependent on S_1. This makes S_7 data dependent on S_1. Similarly, the fact that S_3 is control dependent on S_2 makes S_7 dependent on S_2, though the resulting dependence cannot be classified as either data or control dependence. It is customary to represent the dependences in a program as a graph, with the data and control dependence represented by different kinds of edges, as shown in Figure 2.1. This graph is called a *Program Dependence Graph (PDG)*. For technical reasons, a synthetic node $START$ is added to the graph and all the statements are made control dependent on this node.

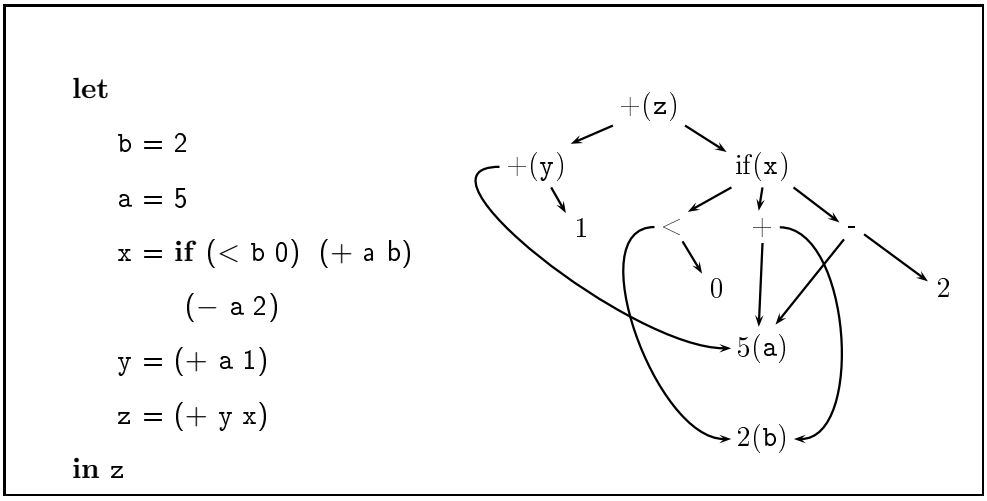


Figure 2.2: Functional program and its tree representation. 2(b) denotes that the node represents a value 2 held through a let binding to a variable b

2.2 Dependences in functional programs

We can define dependences for functional programs in a similar way. Since functional programs do not have assignments of values to variables, the dependences are between expressions instead of statements. Dependence in the context of functional programs could be described as: Given an expression e in a program, what other expressions in the program does the value of e depend on? While we shall formally describe the language for which we propose our analysis, for now we assume the reader's familiarity with a Scheme like language. Consider the earlier imperative program now written as a `let` expression. This can be viewed as a tree as shown in Figure 2.2 with some of the nodes labeled by the variables of `let`. In the absence of function calls, dependence analysis as defined above would be very easy to compute—an expression depends on each of its sub-expressions.

In the context of functional programs, however, the definition can be generalized and made more interesting, especially when the value computed by the program is an algebraic data type. Most functional languages have a set of pre-defined algebraic datatypes (lists, for example) and additionally provide features for users to define their own algebraic datatypes. Values of an algebraic datatype are constructed with *constructors* for example (**cons**, **nil**) and de-constructed with *selectors* (**car** , **cdr**).

Figure 2.3 shows a program fragment computing a structure. As a generalization, we may be interested in knowing the dependences of a "part" of the value of an expression instead of the entire value of the expression. The part that we may be interested in is

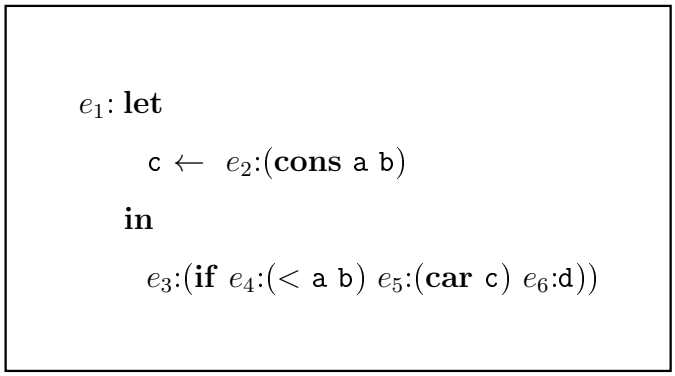


Figure 2.3: A functional program evaluating to a structure. Labels such as e are used to refer to expressions in the ensuing discussion and are not part of the language.

specified through a set of paths, each path representing a composition of selector functions. As an example, assume that we are interested in the part of (the value of) `e1` corresponding to the composition of selectors {**car**∘**cdr**} of the value of `e1`. It is convenient to write this as a set of paths, each path consisting of a sequence of selectors in the order in which these would be applied. In this case the set would consist of a single path **cdr car** (**cdr** followed by **car**). Each prefix of a path represents the root of interest of some sub-structure of `e1`. They are, for this example, the root of `e1` itself represented by the prefix ϵ (the empty prefix of **cdr car**), the root of the structure obtained after a **cdr** selection on `e1` (the prefix **cdr** of **cdr car**), and a **cdr** selection followed by a **car** selection (corresponding to the entire path **cdr car**).

Given that we are interested in {**cdr car**} of `e1`, let us see what sub-structures of other expressions does this depend on. `e1`. Since the value of `e1` is also the value of `e3`, {**cdr car**} of `e1` would obviously depend on {**cdr car**} of `e3`. Further, the value of `e3` is decided by the condition `e4` of the **if**. Therefore there is a control dependence of `e3` (and therefore of `e1`) on {ϵ} of `e4`.[1] Going further down the **if**, Since the value of `e3` is one of `e5` and `d`, {**cdr car**} of `e1` is also dependent {**cdr car**} of `e5` and `d`. Also observe that since the specified part of `e1` depends on {**cdr car**} of `e5`, and `e5` happens to be `(car c)`, the dependence of `e1` on `c` is the set {**car cdr car**}. `c`, a **car** selection `c` to be `(cons a b)`, {**car cdr car**} part of `c` translates to {**cdr car**}. Observe that in obtaining this dependence, we have used the rule (**car** (**cons** x y)) = x of constructor-selector interaction. More importantly, this rule also leads to the conclusion that the

[1]Since `e4` is a scalar, its value is represented by the root ϵ.

specified part of `e1` is not dependent on `b`, and we would not have been able to obtain this precision without incorporating constructor-selector interaction in our analysis.

However, incorporating such rules in the analysis is not simple. In the example, the selector application immediately followed the pair creation. Therefore, it was easy to observe the fact that the result of the application was `a`. In general this may not be the case, the selector and constructor can be widely separated—in fact, they could even be in different functions. Another thing that complicates identifying this selector-construction interaction is the fact that the actual pair that takes part in the selection might take part in further constructions and subsequent selections. For example, in the expression (**car** (**car** (**cons** (**cons** `a` `b`) (**cons** `b` `a`))), the analysis has to correctly identify that the outermost **car** interacts with the constructor (**cons** `a` `b`). Therefore, any analysis which can precisely compute such dependences must contain the identity (**car** (**cons** x y)) $= x$ as its part.

Let e be an expression which evaluates to an algebraic datatype and σ denote a part of this structure. The notion of dependence that we wish to address can now be generalized to: Given the part σ of an expression e, what parts σ_i of other expressions e_i decide the value of σ part of the value of e? One can easily see the applicability of this notion of dependence. For example, in the case of slicing, one can focus on the seemingly erroneous part of the value of an expression and ask what (e_i, σ_i)s does it depend on. If it does not depend on any part of an expression e_i, it can be sliced out. We call the parts of interest σ_i as *demands*. In the following sections, we describe an analysis which incorporates constructor-selector interaction and computes precise dependence information.

2.3 Syntax

Figure 2.4 shows the syntax of our language. We assume familiarity with the basic features of a Scheme-like language. A program in our language is a collection of function definitions followed by a main expression denoted as $e_{\mathbf{main}}$ which for our purposes will always be (**main**). Applications (denoted by the syntactic category *App*) consist of functions or operators applied to variables. Constants are regarded as 0-ary functions. Expressions (*Expr*) are either an **if** expression, a **let** expression that evaluates an application and binds the result to a variable, or a **return** expression. The **return** keyword is used to mark a

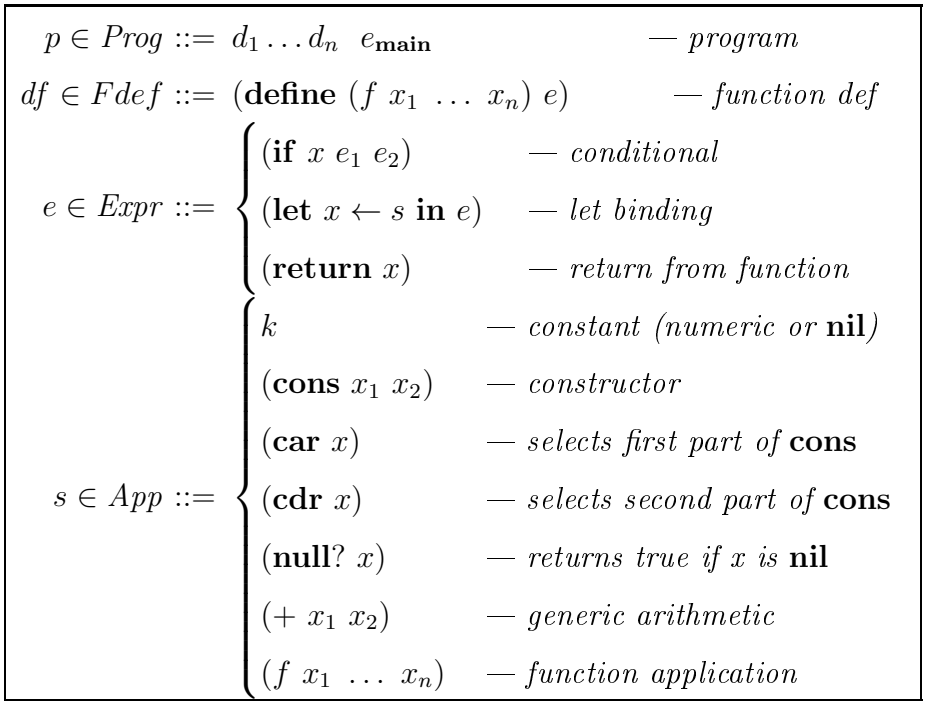


Figure 2.4: The syntax of our language

variable in a returning context of a function. Notable omissions are **lambda** expressions and a provision for user-defined algebraic datatypes.

For ease of presentation, we restrict the language to Administrative Normal Form (ANF) [83]. In this form, the arguments to functions can only be variables. This restriction does not affect expressibility, but has an important notational advantage. As we shall see later, the semantics that we shall ascribe to this language is a generalization of lazy semantics, and every application that is associated with a **let** definition can also be seen as an explication of closure creation. We assume that **let**s in our language are non-recursive—in the expression **let** $x \leftarrow s$ **in** e, x should not occur in s. The restriction of **let** to a single definition is for ease of exposition—generalization to multiple definitions does not add conceptual difficulties. To avoid dealing with scope-shadowing, we assume that all variables in a program are distinct. Neither of these two restrictions affect the expressibility of our language. In fact, in our implementation, we translate a pure subset of Scheme to our language, and, in the slicing tool, map the sliced program back to Scheme. To refer to an expression e, we may annotate it with a label π as $\pi : e$; however the label is not part of the language. To keep the description simple, we shall assume that each program has its own unique set of labels. In other words, a label identifies both the

program point and the program that contains it. We denote the body of a function f as e_f. We assume that each program has a distinguished expression, $e_{\mathbf{main}}$, and the program begins execution with the evaluation of **main**.

2.4 Dependence analysis as propagation of demands

Recall from Section 2.1 that dependence analysis of a functional program, as we wish to view it, answers the following question: *Given that we are interested in a specific part of the structure representing the value of an expression, what parts of other expressions does this value depend on?* The substructure of interest can be identified by a set, whose elements are compositions of selector functions. Each such composition is a sequence of dereferences of **cons** cells through its **car** and **cdr** fields, and ends in the root of some sub-structure. We view this sequence of dereferences as a path from the root of the structure. We introduce the notations **0** to represent dereferencing using the **car** field and **1** for dereferencing using **cdr**. As an example, consider the structure created by the expression (**cons** (**cons** a b) c). The set $\{\mathbf{00}, \mathbf{01}\}$ represents dereferencing paths to the root of various sub-structures of the value of this expressions. This is shown in Figure 2.5. Each element in this set is called an *access path* and the entire set $\{\mathbf{00}, \mathbf{01}\}$ is called a *demand*.

A demand on an expression e represents parts of its value that the context of e *may* be interested in. Here, by context we mean all future computations that may make use of e. Since these parts can be represented by paths from the root of the value, a demand is represented by a set of strings over $(\mathbf{0} + \mathbf{1})^*$. As an example, a demand of $\{\mathbf{10}\}$ on the expression (**cons** x y) means its context *may* be interested in the **car** field of y. This is represented by the prefix **1** of the string **10** in the demand. The absence of the string **0** as a prefix of any string in the demand, on the other hand, indicates that x is *definitely not* of interest. Notice that computation of $\{\mathbf{10}\}$ of (**cons** x y), requires $\{\mathbf{0}\}$ of y. Thus we may think of dependency analysis as computation of a *demand transformer* transforming (or propagating) the demand $\{\mathbf{10}\}$ on (**cons** x y) to demands on its parts—$\{\mathbf{0}\}$ on y and the empty demand (represented by $\emptyset$) on x. As one more example, $\mathbf{1}^*$ in which the context is interested in the spine of a list. The **length** function would represent such a demand on its argument. Similarly if e evaluates to a list, then the demand $\{\mathbf{0}, \mathbf{10}, \mathbf{110}\}$ means that the context may only refer up to the third element of e.

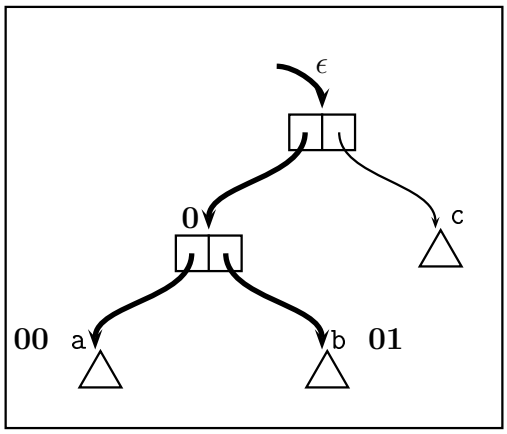


Figure 2.5: Access paths corresponding to the structure (**cons** (**cons** a b) c). Paths corresponding to demand $\{\mathbf{00}, \mathbf{01}\}$ are shown in bold.

The dependence analysis problem is now modeled as follows: Given a demand σ on e, we would like to find the demand σ_i on each of the expressions e_i in the program. Thus dependence analysis lies in computing a demand transformer that, given a demand on e, computes a *demand environment*—a mapping of each expression (represented by its program point π) to its demand. While in the general formulation e can be any expression in the program, in the restricted formulation which suffices for our applications of garbage collection and slicing, e is specifically a call to the main function **main**.

We use σ to represent demands and α to represent access paths. Given two access paths α_1 and α_2, we use the juxtaposition $\alpha_1\alpha_2$ to denote their concatenation. We extend this notation to a concatenate a pair of demands and even to the concatenation of a symbol with a demand: $\sigma_1\sigma_2$ denotes the demand $\{\alpha_1\alpha_2 \mid \alpha_1 \in \sigma \text{ and } \alpha_2 \in \sigma_2\}$ and $\mathbf{0}\sigma$ is, through abuse of notation, a shorthand for $\{\mathbf{0}\alpha \mid \alpha \in \sigma\}$. Also, note that $\sigma_1\sigma_2$, when $\sigma_2 = \emptyset$, is $\emptyset$.

2.4.1 Specification of dependence analysis

We now formally specify the dependence analysis problem. To do this, we first define a non-standard small-step operational semantics called *Demand Guided Semantics (DGS)* that serves as a specification of the real demand on each expression of the program that is required to satisfy the given demand on a designated expression. While the expression of concern can be any expression in the program, for the applications considered in this thesis, namely liveness-based garbage collection and slicing, we shall consider this to be the function **main**, and the user supplied demand will be denoted $\sigma_{\mathbf{main}}$.

```
Function evalAndPrint(expr)
    Data: expr is the expression being evaluated
    val ← evalToWHNF(expr)
    if (pair?(val)) then
        Display "("
        evalAndPrint(car(val))
        Display "."
        evalAndPrint(cdr(val))
        Display ")"
    else
        Display val
```

Algorithm 1: evalAndPrint function that drives the evaluation in a lazy language.

Starting with the demand $\sigma_{\textbf{main}}$ on (**main**), DGS propagates demands to each expression as it is being evaluated during execution. We call the demand thus propagated to an occurrence of an expression as the *dynamic demand* and denote it as δ. A key aspect of DGS is that an expression is evaluated only if the dynamic demand on it is not $\emptyset$. Given an expression and a program point of an arbitrary program, any algorithm which captures the union of the dynamic demands on all occurrence of this expression in the trace of any DGS execution of the program through static analysis is deemed to be a solution of the dependence analysis problem.

Demand Guided Semantics

The execution state of Demand Guided Semantics has the dynamic demand as a component. The evaluation aspect of Demand Guided Semantics can be thought of as a generalization of lazy semantics. In lazy semantics, the evaluation of the user supplied main program **main** is mediated through a runtime support, which we can model as a function called **evalAndPrint**. **evalAndPrint** evaluates **main** till it reaches Weak Head Normal Form (WHNF) [74]. Since our language is first order, this means that an expression has to be evaluated to its value and printed in the case of a base type, and to a partially evaluated expression with the outer constructor (**cons**) exposed, in the case of a pair or a list. **evalAndPrint** then recursively evaluates and prints the (possibly)

```
Function evalAndPrint(expr, δ)
    Data: expr is the expression to be evaluated
    Data: δ is the demand on expr
    if (¬(δ == ∅)) then
        val ← evalToWHNF(expr)
    if (pair?(val)) then
        Display "("
        evalAndPrint(car(val), δ1)          // δ1 = {α | 0α ∈ δ}
        Display "."
        evalAndPrint(cdr(val), δ2)          // δ2 = {β | 1β ∈ δ}
        Display ")"
    else
        Display val
```

Algorithm 2: Modified **evalAndPrint** function to drive evaluation in demand guided semantics.

unevaluated expressions constituting the head and the tail of **main**, printing appropriate delimiters in between. This is shown in Figure 1. We wish to point out that:

1. The reason why a program in a lazy language is evaluated in this way is to reconcile two requirements. First, lazy semantics dictates that the evaluation of *any* expression in the program means 'evaluation till WHNF'. However, the top level expression must be fully evaluated for the user to see the results. Therefore, each expression is evaluated by default till WHNF. However, the top-level expression **main** is guided to a full evaluation by **evalAndPrint**.
2. This strategy effectively puts a demand of $(\mathbf{0} + \mathbf{1})^*$ (full evaluation) on the main expression.

In DGS, we generalize the notion of lazy evaluation, so that instead of evaluating the main expression fully, it is evaluated to the extent given by an arbitrary demand δ. As examples, if e is a list and δ is 1^*, then e is evaluated to expose its entire spine, the individual elements of the list can remain unevaluated. The modified **evalAndPrint** function for a demand guided evaluation is shown in Figure 2.

Premise	Transition	Rule name
δ is $\emptyset$	$\rho, (\rho', y, e', \delta') : S, e, \delta \rightsquigarrow \rho', S, e', \delta'$	NO-EVAL
	$\rho, (\rho', y, e, \delta') : S, \kappa, \delta \rightsquigarrow \rho' \oplus \{y \mapsto v\}, S, e, \delta'$	CONST
$\rho(x)$ is $\langle s, \rho' \rangle$	$\rho, S, x, \delta \rightsquigarrow \rho', S, s, \delta$	VAR
$\rho(x)$ is $\langle (\mathbf{id}\ y), \rho' \rangle$	$\rho, S, x, \delta \rightsquigarrow \rho', S, y, \delta$	ID
	$\rho, S, (\mathbf{car}\ x), \delta \rightsquigarrow \rho, S, x, \mathbf{0}\delta$	CAR
	$\rho, S, (\mathbf{cdr}\ x), \delta \rightsquigarrow \rho, S, x, \mathbf{1}\delta$	CDR
	$\rho, (\rho', w, e, \delta') : S, (\mathbf{cons}\ x\ y), \epsilon \rightsquigarrow$ $\rho' \oplus \{w \mapsto (\langle (\mathbf{id}\ x), \rho \rangle, \langle (\mathbf{id}\ y), \rho \rangle)\}, S, e, \delta'$	CONS
$\delta' = \{\alpha \mid \mathbf{0}\alpha \in \delta\}$	$\rho, S, (\mathbf{cons}\ x\ y), \delta \rightsquigarrow \rho, S, x, \delta'$	CAR-CONS
$\delta' = \{\alpha \mid \mathbf{1}\alpha \in \delta\}$	$\rho, S, (\mathbf{cons}\ x\ y), \delta \rightsquigarrow \rho, S, x, \delta'$	CDR-CONS
$\rho(x), \rho(y) \in \mathbb{N}$	$\rho, (\rho', z, e, \delta') : S, (+\ x\ y), \delta \rightsquigarrow$ $\rho' \oplus \{z \mapsto (+\ \rho(x)\ \rho(y))\}, S, e, \delta'$	PRIM-ADD
$\rho(x)$ is $\langle s, \rho' \rangle$	$\rho, S, (+\ x\ y), \delta \rightsquigarrow \rho, (\rho, x, (+\ x\ y), \delta) : S, x, \epsilon$	PRIM-1-CLO
$\rho(y)$ is $\langle s, \rho' \rangle$	$\rho, S, (+\ x\ y), \delta \rightsquigarrow \rho, (\rho, y, (+\ x\ y), \delta) : S, y, \epsilon$	PRIM-2-CLO
f defined as $(\mathbf{define}\ (f\ \vec{y})\ e_f)$	$\rho, S, (f\ \vec{x}), \delta \rightsquigarrow [\vec{y} \mapsto \langle (\mathbf{id}\ \vec{x}), \rho \rangle], S, e_f, \delta$	FUNCALL
	$\rho, S, (\mathbf{let}\ x \leftarrow s\ \mathbf{in}\ e), \delta \rightsquigarrow \rho \oplus \{x \mapsto \langle s, \rho \rangle\}, S, e, \delta$	LET
$\rho(x) \in \mathbb{N}\ \&\ \rho(x) \neq 0$	$\rho, S, (\mathbf{if}\ x\ e_1\ e_2), \delta \rightsquigarrow \rho, S, e_1, \delta$	IF-TRUE
$\rho(x) \in \mathbb{N}\ \&\ \rho(x) = 0$	$\rho, S, (\mathbf{if}\ x\ e_1\ e_2), \delta \rightsquigarrow \rho, S, e_2, \delta$	IF-FALSE
$\rho(x)$ is $\langle s, \rho' \rangle$	$\rho, S, (\mathbf{if}\ x\ e_1\ e_2), \delta \rightsquigarrow \rho, (\rho, x, (\mathbf{if}\ x\ e_1\ e_2), \delta) : S, x, \epsilon$	IF-CLO
$\rho(x)$ is $\langle s, \rho' \rangle$	$\rho, S, (\mathbf{return}\ x), \delta \rightsquigarrow \rho,\ S, x, \delta$	RETURN-CLO

Figure 2.6: Demand guided execution semantics. NO-EVAL has precedence over all rules.

We now specify the domains used by the semantics:

$$
\begin{aligned}
d : \mathit{Data} &= \mathit{Val} + \mathit{Clo} && \text{– Values \& Closures} \\
v : \mathit{Val} &= \mathbb{N} + \{\mathbf{nil}\} + \mathit{Data} \times \mathit{Data} && \text{– Values} \\
c : \mathit{Clo} &= (\mathit{App} \times \mathit{Env}) && \text{– Closures} \\
\rho : \mathit{Env} &= \mathit{Var} \to \mathit{Data} && \text{– Environment}
\end{aligned}
$$

A data value d may either be an evaluated value, denoted by v, or a closure. A closure is a pair $\langle s, \rho \rangle$ in which s is an unevaluated application, and ρ maps free variables of s to data values. An environment is a mapping from the set of variables of the program *Var* to *Data*. The notation $[\vec{y} \mapsto \rho(\vec{x})]$ represents an environment that maps the formal

arguments y_i to the bindings of the actual arguments x_i. $\rho \oplus \rho'$ represents the environment ρ shadowed by ρ' and $\lfloor \rho \rfloor_X$ represents the environment restricted to the variables in the set X. Finally $FV(s)$ represents the free variables in the application s.

The DGS of our language is shown in Figure 2.6. The semantics of expressions and applications are given by transitions of the form $\rho, S, e, \delta \rightsquigarrow \rho', S', e', \delta'$. Here ρ is an environment that maps variables to their bindings, S is a stack of continuation frames, e is the current expression being evaluated, and δ is the demand on e. Each continuation frame is a 4-tuple $(\rho, x, e_{next}, \delta)$, signifying that the variable x has to be updated with the value of the currently evaluating expression and e_{next} is the next expression to be evaluated in the environment ρ with the demand on e_{next} being δ. The initial state of the transition system is: $([\ ]_\rho, [\,([\ ]_\rho, \textbf{ans}, (\textbf{evalAndPrint}), \delta_{\textbf{main}})\,], (\textbf{main}), \{\epsilon\})$ in which $[\ ]_\rho$ is the empty environment. The initial stack consists of a single continuation frame in which **ans** is a distinguished variable that will eventually be updated with the value of (**main**) and (**evalAndPrint**) will be picked next for execution. The function **evalAndPrint** halts the program if the value of (**main**) is already fully evaluated, else it first produces a DGS trace starting from the state $([\ ]_\rho, [\,([\ ]_\rho, \textbf{ans}, (\textbf{evalAndPrint}), \delta_1)\,], (\textbf{car main}), \epsilon)$ where $\delta_1 = \{\alpha \mid \textbf{0}\alpha \in \delta\}$, followed by the DGS trace starting from $([\ ]_\rho, [\,([\ ]_\rho, \textbf{ans}, (\textbf{evalAndPrint}), \delta_2)\,], (\textbf{cdr main}), \epsilon)$ where $\delta_2 = \{\beta \mid \textbf{1}\beta \in \delta\}$.

In the demand guided semantics shown in Figure 2.6, evaluation of a let expression (**let** $x \leftarrow s$ **in** e) does not result in the evaluation of s. Instead, as the LET rule shows, a closure is created and bound to x. While evaluating a function body, evaluation of closures is initially triggered while checking an **if** condition (IF-CLO) or at a **return** (RETURN-CLO). This, in turn, may trigger evaluation of more closures. As an example of closure evaluation, we explain the rules for **car** and **cons**. If the demand δ on (**car** x) is $\emptyset$ then it is not evaluated at all (NO-EVAL). If δ is non-null, then x is evaluated with the propagated demand $\textbf{0}\delta$. In a well typed program, the evaluation of x should result in a closure say (**cons** y z). The DGS semantics now uses the CAR-CONS rule to select y for evaluation with the propagated demand obtained by stripping off leading **0** from all strings in $\textbf{0}\delta$. This gives back δ as the demand to be propagated to y. However, if the surrounding context of (**cons** y z) had been a **cdr** instead of **car**, then this would have resulted in $\emptyset$ and then y would not have been evaluated any further (NO-EVAL). Also notice that the rule for function calls is defined through the use of the identity function

id. We have introduced this due to purely technical reasons. The evaluation of (**id** x) is defined by the rule ID, it results in the evaluation of x in the same execution context. Its introduction simplifies the definition of subsumption in Section 3.3.

The set operation $\{\alpha \,|\, \mathbf{0}\alpha \in \delta\}$ can be described algebraically by introducing the symbol $\bar{\mathbf{0}}$ defined as $\bar{\mathbf{0}}\delta = \{\alpha \,|\, \mathbf{0}\alpha \in \delta\}$ with the derived property that $\bar{\mathbf{0}}\mathbf{0}$ rewrites to ϵ. Similarly, we define $\bar{\mathbf{1}}\delta = \{\alpha \,|\, \bar{\mathbf{0}}\alpha \in \delta\}$ with the derived property that $\bar{\mathbf{1}}\mathbf{1}$ rewrites to ϵ. We now formally define the dependence analysis problem as follows:

Definition 2.1 *The dependence analysis problem is to find an algorithm* $\mathbb{A}$ *such that given a program* P*, a demand* δ*, a control point* π*, and a string* $w \in (\mathbf{0}+\mathbf{1})^*$ *will answer* yes *if there exists a DGS trace of* P *and* δ *in which the expression at* π *appears with a dynamic demand* δ' *containing* w*, and* no *otherwise.*

We introduce a predicate $\mathsf{prop}(e, \delta, \pi : e', \delta')$ to denote that there exists a DGS trace of an expression e with demand δ such that the expression e' at the program point π appears on the trace with a dynamic demand δ'. Thus the dependence analysis problem is to find an algorithm $\mathbb{A}$ such that $\forall P \forall \delta \forall \pi \forall w. \mathbb{A}(P, \delta, \pi, w) = \exists \delta'. \mathsf{prop}(P, \delta, \pi, \delta')$, and $w \in \delta'$. We are now ready to prove a result that shows that dependence analysis is undecidable.

2.4.2 Undecidability of dependence analysis

We use the symbol w to denote strings in $(\mathbf{0}+\mathbf{1})^*$, α and β to denote strings in $(\mathbf{0}+\mathbf{1}+\bar{\mathbf{0}}+\bar{\mathbf{1}})^*$, and γ to denote strings of grammar symbols, i.e. strings of non-terminals and terminals. We also name the set consisting of the two non-context-free productions $\bar{\mathbf{0}}\mathbf{0} \to \epsilon$ and $\bar{\mathbf{1}}\mathbf{1} \to \epsilon$ as unrestricted. We first show that for a class of grammars CG consisting of a set of context-free productions over the terminal symbols $\{\mathbf{0}, \mathbf{1}, \bar{\mathbf{0}}, \bar{\mathbf{1}}\}$ along with the fixed set of non-context-free productions unrestricted, the problem of whether ϵ belongs to an arbitrary grammar in the class is undecidable. We then consider a subset of CG, say CG', that is large enough to replay the Undecidability proof. Specifically, the set CG' corresponds to the set of all Turing Machines. Finally we show that any grammar G in CG' can be converted to a program P such that the problem of whether ϵ belongs to $\mathscr{L}(G)$ can be reduced to the dependence analysis problem of P.

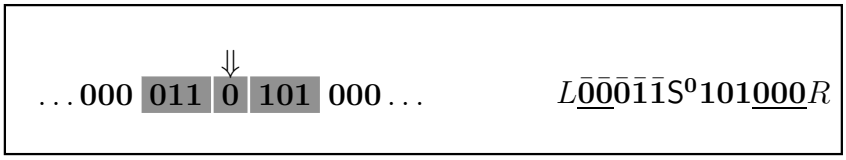


Figure 2.7: Figure illustrating the correspondence between TM state and grammar sentential form. Shaded part represents the region of interest and ⇓ represents the location of the TM head. Underlined symbols are spurious symbols produced by the L and R productions that can be erased later by S^c_{final}.

Lemma 2.2 *Consider the class of grammars CG in which each grammar G is of the kind $(N,\ \{\mathbf{0},\ \mathbf{1},\ \bar{\mathbf{0}},\ \bar{\mathbf{1}}\},\ p \cup \mathsf{unrestricted},\ S)$. Here N is a set of non-terminals and p is a set of context-free productions containing the distinguished production $S \to \gamma$, where γ is a string of grammar symbols that does not contain S. The problem of whether an arbitrary grammar G in this set recognizes ϵ is undecidable.*

Proof. We reduce the Halting problem to the ϵ-recognition problem of grammars in CG. We assume that the Turing Machine (TM) is deterministic, the input w to the Turing Machine is a unary string in $\mathbf{1}^*$ and the blank symbol is represented by $\mathbf{0}$. We shall represent a TM configuration as $w_l(\mathsf{S}, c)w_r$, where w_l and w_r are regions of the tape that have either been visited or contain the symbol $\mathbf{1}$, c is the symbol under the head and S is the current state of the TM. We shall call $w_l c w_r$ as the *region of interest* in the tape. We construct a grammar G such that the machine will halt on w if and only if $\epsilon \in \mathscr{L}(G)$. The grammar will have the following productions:

1. *Fixed productions:* These are the productions in $\mathsf{unrestricted}$ and the productions $L \to L\,\bar{\mathbf{0}} \mid \epsilon$ and $R \to \mathbf{0}R \mid \epsilon$.
2. *Productions related to TM transitions:* For each combination of state and symbol (S, c), the grammar will contain the non-terminal S^c. Correspondences between the moves of the TM and the grammar productions are as follows:
   (a) For each transition $(\mathsf{S}_i, c) \to (\mathsf{S}_j, c', L)$, there are two productions $\mathsf{S}^c_i \to \mathbf{0}\mathsf{S}^{\mathbf{0}}_j c'$ and $\mathsf{S}^c_i \to \mathbf{1}\mathsf{S}^{\mathbf{1}}_j c'$.
   (b) For each transition $(\mathsf{S}_i, c) \to (\mathsf{S}_j, c', R)$, there are two productions $\mathsf{S}^c_i \to \overline{c'}\mathsf{S}^{\mathbf{0}}_j\bar{\mathbf{0}}$ and $\mathsf{S}^c_i \to \overline{c'}\mathsf{S}^{\mathbf{1}}_j\bar{\mathbf{1}}$.
3. *Productions related to the final state:* For every symbol c, there is a non-terminal S^c_{final} where S_{final} is the final state of the TM. We add the productions $\mathsf{S}^c_{final} \to \mathbf{0}\mathsf{S}^c_{final}$,

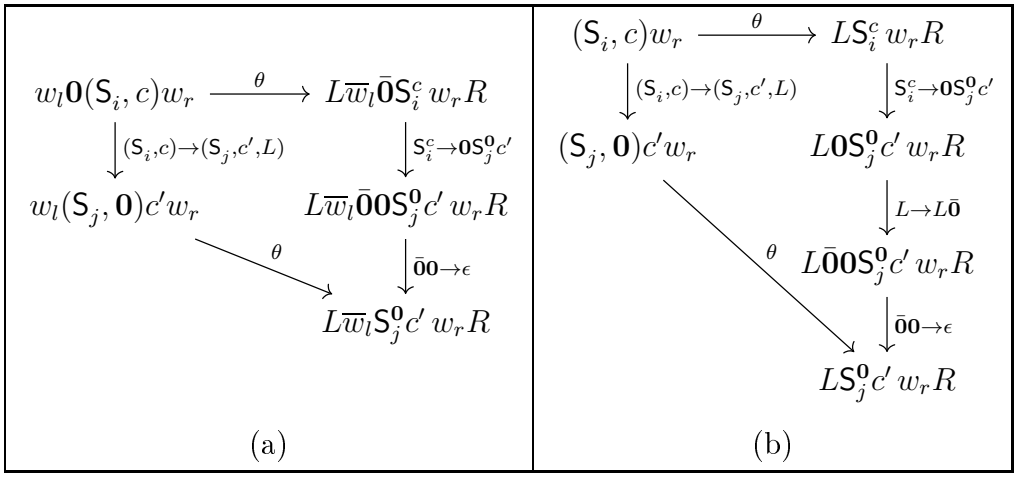


Figure 2.8: Commutative diagrams illustrating the invariant θ mapping TM moves to sentential forms.

$\mathsf{S}^c_{final} \to \mathbf{1}\mathsf{S}^c_{final}$, $\mathsf{S}^c_{final} \to \mathsf{S}^c_{final}\bar{\mathbf{0}}$, $\mathsf{S}^c_{final} \to \mathsf{S}^c_{final}\bar{\mathbf{1}}$ and $\mathsf{S}^c_{final} \to \epsilon$.

4. *Production related to the start state:* Assume that the TM starts in a state S_{init} with an input w and the head positioned to the immediate left of w. Then there is a production $\mathsf{S} \to L\mathsf{S}^0_{init}wR$, where S would be regarded as the start symbol of the grammar.

We now first show that if the TM halts on w, then there is a derivation $\mathsf{S} \stackrel{*}{\Rightarrow} \epsilon$. To do this, we define a mapping θ that serves as an invariant relation from configurations of the TM to sentential forms.

> θ maps a TM configuration $w_l(\mathsf{S}, c)w_r$ to the sentential form $L\overline{w}_l\mathsf{S}^c\, w_rR$, where $\overline{w}_l$ is the same as w_l but with each tape symbol c in w_l replaced by $\overline{c}$. The L and the R non-terminals act as markers that delimit the infinite tape to its region of interest.

Thus, if $\mathsf{S}^0_{init}w$ is the initial configuration of the TM, then the matching sentential form is $L\overline{w}_l\mathsf{S}^0_{init}\, w_rR$, which can be derived in a single step from S. For any move of the TM, we now specify the sequence of derivation steps that would maintain the invariant.

1. Assume that the TM moves left using the transition rule $(\mathsf{S}_i, c) \to (\mathsf{S}_j, c', L)$. There are two sub cases:
   (a) If the current configuration of the TM is $w_l\mathbf{0}(\mathsf{S}_i, c)w_r$, then θ defines the current sentential form to be $L\overline{w}_l\bar{\mathbf{0}}\mathsf{S}^c_i\, w_rR$. The corresponding derivation first uses the production $\mathsf{S}^c_i \to \mathbf{0}\mathsf{S}^{\mathbf{0}}_j c'$ and follows it up using $\bar{\mathbf{0}}\mathbf{0} \to \epsilon$. This is shown in Figure 2.8(a). Notice that the invariant continues to be maintained between the new

state of the TM and the sentential form at the end of these two derivation steps. Similarly, if the TM configuration is $w_l\mathbf{1}(\mathsf{S}_i, c)w_r$ and the TM transition remains the same, then the corresponding derivation first uses the production $\mathsf{S}_i^c \rightarrow \mathbf{1}\mathsf{S}_j^{\mathbf{1}}c'$ and then simplifies using $\bar{\mathbf{1}}\mathbf{1} \rightarrow \epsilon$.

(b) Let the current configuration of the TM be $(\mathsf{S}_i, c)w_r$. Then θ defines the current sentential form as $L\mathsf{S}_i^c\, w_r R$. The corresponding steps in the derivation are: first move the left marker using the production $L \rightarrow L\bar{\mathbf{0}}$, expand S_i^c using the $\mathsf{S}_i^c \rightarrow \mathbf{0}\mathsf{S}_j^{\mathbf{0}}c'$ and simplify using $\bar{\mathbf{0}}\mathbf{0} \rightarrow \epsilon$. Figure 2.8(b) shows that the invariant continues to be maintained.

2. Now assume that the TM makes a right move using the transition rule $(\mathsf{S}_i, c) \rightarrow (\mathsf{S}_j, c', R)$. There are again two sub cases:

(a) If the current configuration of the TM is $w_l(\mathsf{S}_i, c)\mathbf{0}w_r$, then the current sentential form is $L\overline{w}_l\mathsf{S}_i^c\,\mathbf{0}w_r R$. The corresponding derivation first uses the production $\mathsf{S}_i^c \rightarrow \overline{c'}\mathsf{S}_j^{\mathbf{0}}\bar{\mathbf{0}}$ and follows it up using $\bar{\mathbf{0}}\mathbf{0} \rightarrow \epsilon$. Similarly, if the TM configuration is $w_l(\mathsf{S}_i, c)\mathbf{1}w_r$, then the corresponding derivation first uses the production $\mathsf{S}_i^c \rightarrow \overline{c'}\mathsf{S}_j^{\mathbf{1}}\bar{\mathbf{1}}$ and then simplifies using $\bar{\mathbf{1}}\mathbf{1} \rightarrow \epsilon$.

(b) Let the current configuration of the TM be $w_l(\mathsf{S}_i, c)$. Then the corresponding steps in the derivation are: first move the right marker using the production $R \rightarrow \mathbf{0}R$, expand S_i^c using the $\mathsf{S}_i^c \rightarrow \overline{c'}\mathsf{S}_j^{\mathbf{0}}\bar{\mathbf{0}}$ and simplify using $\bar{\mathbf{0}}\mathbf{0} \rightarrow \epsilon$. It is easy to verify that in both the sub cases of 2, the invariant continues to be maintained.

The idea behind the productions is explained with an example: Assume that the traversed part of the TM is $\mathbf{01}(\mathsf{S}_i, \mathbf{0})\mathbf{00}$ and therefore the current sentential form is $L\bar{\mathbf{0}}\bar{\mathbf{1}}\mathsf{S}_i^{\mathbf{0}}\mathbf{00}R$. Also assume that the TM has a transition $(\mathsf{S}_i, \mathbf{0}) \rightarrow (\mathsf{S}_j, \mathbf{1}, L)$. Since the next corresponding step in the derivation has to be done without any prior knowledge of whether the symbol to the left of the tape is a $\mathbf{0}$ or a $\mathbf{1}$, two productions are provided, and the invariant will be maintained only if the production $\mathsf{S}_i^{\mathbf{0}} \rightarrow \mathbf{1}\mathsf{S}_j^{\mathbf{1}}\mathbf{1}$ is chosen for the next step in the derivation. This gives the configuration $L\bar{\mathbf{0}}\bar{\mathbf{1}}\mathbf{1}\mathsf{S}_j^{\mathbf{1}}\mathbf{100}R$. Simplification with the production $\bar{\mathbf{1}}\mathbf{1} \rightarrow \epsilon$ yields $L\bar{\mathbf{0}}\mathsf{S}_j^{\mathbf{1}}\mathbf{100}R$, which exactly corresponds to the changed configuration of the TM.

When the TM comes to a halt in a configuration $w_l\mathsf{S}_{final}^c\, w_r$, the corresponding sentential form is $L\overline{w}_l\mathsf{S}_{final}^c\, w_r R$. In the subsequent derivations both L and R derive ϵ and the S_{final}^c productions are used to generate symbols that can be used by the productions

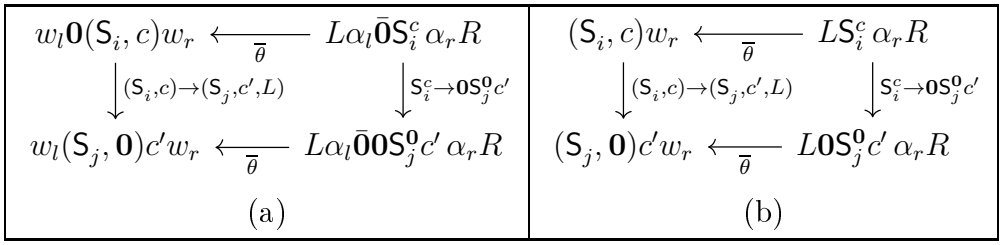


Figure 2.9: Commutative diagrams illustrating the invariant $\overline{\theta}$ mapping sentential forms to TM moves.

in unrestricted to erase $\overline{w_l}$ and w_r. The derivation ends with S_{final}^c deriving ϵ. Clearly if the TM halts on the input string, then there is a derivation $\mathsf{S} \stackrel{*}{\Rightarrow} \epsilon$.

Before proving the converse, we state a property of derivations in the constructed grammar. The productions used for derivations can be categorized as (1) productions with S_i^c on the LHS, (2) productions with L or R on the LHS, and (3) productions in unrestricted.

Lemma 2.3 *Consider a derivation $\mathsf{S} \stackrel{*}{\Rightarrow} \gamma$ in which productions are applied in some sequence. The following pairs of consecutive productions in the derivation can be interchanged:*

1. $\mathsf{S}_i^c \to \gamma_1$ $L \to \gamma_2$ *and* $L \to \gamma_2$ $\mathsf{S}_i^c \to \gamma_1$

2. $\mathsf{S}_i^c \to \gamma_1$ $R \to \gamma_2$ *and* $R \to \gamma_2$ $\mathsf{S}_i^c \to \gamma_1$

3. $L \to \gamma_1$ $R \to \gamma_2$ *and* $R \to \gamma_2$ $L \to \gamma_1$

Also, the pair of consecutive productions, $\bar{\mathbf{0}}\mathbf{0} \to \epsilon$ $X \to \gamma_1$ *can be replaced by* $X \to \gamma_1$ $\bar{\mathbf{0}}\mathbf{0} \to \epsilon$ *where X is one of S_i^c, L or R. Similarly,* $\bar{\mathbf{1}}\mathbf{1} \to \epsilon$ $X \to \gamma_1$ *can be replaced by* $X \to \gamma_1$ $\bar{\mathbf{1}}\mathbf{1} \to \epsilon$.

As a consequence of Lemma 2.3, if $\mathsf{S} \stackrel{*}{\Rightarrow} \gamma$ through a sequence of productions, we can derive γ through an alternate derivation that re-orders the sequence by applying the S_i^c productions first, followed by the L and R productions and finally the unrestricted productions. Notice that the new derivation must retain the order of productions in the same category.

Also notice another property of sentential forms. A $\bar{\mathbf{0}}$ (or $\bar{\mathbf{1}}$) can *only* be cancelled by a $\mathbf{0}$ (or $\mathbf{1}$) on its immediate right. Similarly, a $\mathbf{0}$ (or $\mathbf{1}$) can *only* be cancelled by a $\bar{\mathbf{0}}$ (or $\bar{\mathbf{1}}$) on its immediate left. Define an *uncancellable-pair* as the string $\bar{\mathbf{0}}\mathbf{1}$ or $\bar{\mathbf{1}}\mathbf{0}$. We then make the following claim:

Claim 2.4 *If a sentential form contains an uncancellable-pair it can never derive ϵ.*

We are now in a position to prove the converse result, that if $\mathsf{S} \rightarrow L\mathsf{S}^0_{init}wR \stackrel{*}{\Rightarrow} \epsilon$, then starting with the configuration $(\mathsf{S}_{init}, c)w$, the TM reaches a final state. Because of Lemma 2.3, we can assume that the productions with S^c_i are employed before L, R or unrestricted. Consider the segment of the derivation that starts with $L\mathsf{S}^0_{init}wR$ and ends with the sentential form that has S^c_{final} for the first time. To derive ϵ there must be such a sentential form. We now specify an invariant $\overline{\theta}$ mapping sentential forms to TM configurations.

> Each sentential form has the structure $L\alpha_l\mathsf{S}^c_i\alpha_rR$, where $\alpha_l, \alpha_r \in (\mathbf{0}+\mathbf{1}+\bar{\mathbf{0}}+\bar{\mathbf{1}})^*$. Given such a sentential form, the corresponding TM state is $w_l(\mathsf{S}_i, c)w_r$, where $\alpha_l \stackrel{*}{\Rightarrow} \beta_l\overline{w_l}$, and $\alpha_r \stackrel{*}{\Rightarrow} w_r\beta_r$ where $\beta_l \in \mathbf{0}^*$ and $\beta_r \in \bar{\mathbf{0}}^*$.

The correspondence between derivation steps and TM moves is as follows:

1. Assume that the production chosen for the next step in the derivation is $\mathsf{S}^c_i \rightarrow \mathbf{0}\mathsf{S}^{\mathbf{0}}_j c'$. This production corresponds to the unique left move $(\mathsf{S}_i, c) \rightarrow (\mathsf{S}_j, c', L)$. To derive ϵ without getting stuck, the current sentential form has to be either $L\alpha_l\bar{\mathbf{0}}\mathsf{S}^c_i\alpha_rR$ or $L\mathsf{S}^c_i\alpha_rR$.
   (a) If the sentential form is $L\alpha_l\bar{\mathbf{0}}\mathsf{S}^c_i\alpha_rR$, then because of the invariant $\overline{\theta}$, the current TM configuration is $w_l\mathbf{0}(\mathsf{S}_i, c)w_r$, where $\alpha_l\bar{\mathbf{0}} \stackrel{*}{\Rightarrow} \beta_l\overline{w_l}\bar{\mathbf{0}}$, $\alpha_r \stackrel{*}{\Rightarrow} w_r\beta_r$, $\beta_l \in \mathbf{0}^*$ and $\beta_r \in \bar{\mathbf{0}}^*$. Clearly the next sentential form is $L\alpha_l\bar{\mathbf{0}}\mathbf{0}\mathsf{S}^0_j c'\alpha_rR$ and the next TM configuration is $w_l(\mathsf{S}_j, \mathbf{0})c'w_r$. The invariant is maintained once again, because $\alpha_l\bar{\mathbf{0}}\mathbf{0} \stackrel{*}{\Rightarrow} \beta_l\overline{w_l}$ and $c'\alpha_r \stackrel{*}{\Rightarrow} c'w_r\beta_r$. This is shown in Figure 2.9(a).
   (b) If the sentential form is $L\mathsf{S}^c_i\alpha_rR$, then the current TM configuration is $(\mathsf{S}_i, c)w_r$, where $\alpha_r \stackrel{*}{\Rightarrow} w_r\beta_r$ and $\beta_r \in \bar{\mathbf{0}}^*$. The next sentential form is $L\mathbf{0}\mathsf{S}^0_j c'\alpha_rR$ and the next TM configuration is $(\mathsf{S}_j, \mathbf{0})c'w_r$. The invariant is maintained because $\mathbf{0} \in \mathbf{0}^*$ and $c'\alpha_r \stackrel{*}{\Rightarrow} c'w_r\beta_r$. This is shown in Figure 2.9(b).
2. If the next production chosen is $\mathsf{S}^c_i \rightarrow \mathbf{1}\mathsf{S}^{\mathbf{1}}_j c'$, then it also corresponds to the left move $(\mathsf{S}_i, c) \rightarrow (\mathsf{S}_j, c', L)$. The current sentential form necessarily has the structure $L\alpha_l\bar{\mathbf{1}}\mathsf{S}^c_i\alpha_rR$ and the corresponding TM configuration is $w_l\mathbf{1}(\mathsf{S}_i, c)w_r$, where $\alpha_l\bar{\mathbf{1}} \stackrel{*}{\Rightarrow} \beta_l\overline{w_l}\bar{\mathbf{1}}$, $\alpha_r \stackrel{*}{\Rightarrow} w_r\beta_r$, $\beta_l \in \mathbf{0}^*$ and $\beta_r \in \bar{\mathbf{0}}^*$. The next sentential form is $L\alpha_l\bar{\mathbf{1}}\mathbf{1}\mathsf{S}^0_j c'\alpha_rR$ and the next TM configuration is $w_l(\mathsf{S}_j, \mathbf{1})c'w_r$. The invariant is maintained once again, because $\alpha_l\bar{\mathbf{1}}\mathbf{1} \stackrel{*}{\Rightarrow} \beta_l\overline{w_l}$ and $c'\alpha_r \stackrel{*}{\Rightarrow} c'w_r\beta_r$.

3. The productions $\mathsf{S}_i^c \rightarrow \overline{c'}\mathsf{S}_j^{\mathbf{0}}\bar{\mathbf{0}}$ or $\mathsf{S}_i^c \rightarrow \overline{c'}\mathsf{S}_j^{\mathbf{1}}\bar{\mathbf{1}}$ correspond to a right move of the TM $(\mathsf{S}_i, c) \rightarrow (\mathsf{S}_j, c', R)$ and can be reasoned similarly.

It is important to notice that a wrong choice of production for the current sentential form will result in a sentential form that will contain an uncancellable-pair and will not be able to derive ϵ.

Since the original derivation derived ϵ, it had to arrive at a sentential form containing S_{final}^c. Therefore, the re-ordered derivation will also reach a sentential form such as $L\alpha_l\mathsf{S}_{final}^c\alpha_r R$. Because of the invariant the TM will be in a configuration $w_l(\mathsf{S}_{final}, c)w_r$ and halt. This completes the reduction. ■

Let us enumerate the kinds of context-free productions used in the proof of Lemma 2.2. They are (i) the productions corresponding to the starting state of the TM, $\mathsf{S} \rightarrow L\mathsf{S}_i wR$, $w \in (\mathbf{0}+\mathbf{1})^*$ (ii) the productions corresponding to the intermediate states, $\mathsf{S}_i \rightarrow \mathbf{0}\mathsf{S}_j c$, $\mathsf{S}_i \rightarrow \mathbf{1}\mathsf{S}_j c$, $\mathsf{S}_i \rightarrow \overline{c}\mathsf{S}_j\bar{\mathbf{0}}$ and $\mathsf{S}_i \rightarrow \overline{c}\mathsf{S}_j\bar{\mathbf{1}}$, where $c \in \{\mathbf{0}, \mathbf{1}\}$, (iii) the productions corresponding to the final state, $\mathsf{S}_i \rightarrow c\mathsf{S}_i$, $\mathsf{S}_i \rightarrow \mathsf{S}_i\overline{c}$ and $\mathsf{S}_i \rightarrow \epsilon$, and (iv) the productions corresponding to L and R. The following lemma is obvious:

Lemma 2.5 *Consider the subclass CG' of grammars $(N, \{\mathbf{0}, \mathbf{1}, \bar{\mathbf{0}}, \bar{\mathbf{1}}\}, p \cup \mathsf{unrestricted}, S)$, in which the productions in p are restricted to the forms described above. The ϵ-recognition problem for CG' is undecidable.*

For grammars in CG', define a canonical derivation as one which first uses the R-productions, then the S-productions, followed by the L-productions, and finally uses the productions in unrestricted. In other words, a canonical derivation is a rightmost derivation $S \stackrel{*}{\Rightarrow}_{rm} \alpha$, followed by the use of productions in unrestricted. We present the following claim without proof:

Claim 2.6 *Consider a grammar in CG'. For every derivation of a string in G, there is also a canonical derivation of the same string.*

We now show how to construct a program P from given a grammar $G \in CG'$, such that the language recognition of G is related to the result of dependence analysis of P. This is shown in Algorithm 3. The reader can verify that for the example productions shown in Figure 2.10, application of **pgm** will result in the program shown alongside. We now have the following result.

Function pgm(P)
Data: P is a production of the form $\mathsf{S} \to \gamma$ or $\mathsf{S} \to \gamma_1 \mid \gamma_2$
Result: F, function corresponding to P
begin
$\mathsf{var} \leftarrow$ **createNewVar**()
switch P **do**
case $\mathsf{S} \to \gamma$ **do**
$F \leftarrow$ (**define** (S var))(**pgm'**(γ, var))
case $\mathsf{S} \to \gamma_1 \mid \gamma_2$ **do**
$F \leftarrow$ (**define** (S var))(**if** $*$ (**pgm'**(γ_1, var)) (**pgm'**(γ_2, var)))
return F
Function pgm'$(\gamma, \mathsf{curvar})$
Data: A string γ of grammar symbols representing the RHS of a production and curvar, the context variable for γ
Result: The program fragment for γ
begin
$\mathsf{var} \leftarrow$ **createNewVar**()
switch γ **do**
case ϵ **do**
(**return** curvar)
case $\mathbf{0}\gamma'$ **do**
(**let** var $\leftarrow$ (**car** curvar) **in pgm'**(γ', var))
case $\mathbf{1}\gamma'$ **do**
(**let** var $\leftarrow$ (**cdr** curvar) **in pgm'**(γ', var))
case $\bar{\mathbf{0}}\gamma'$ **do**
(**let** var $\leftarrow$ (**cons** curvar _) **in pgm'**(γ', var))
case $\bar{\mathbf{1}}\gamma'$ **do**
(**let** var $\leftarrow$ (**cons** _ curvar) **in pgm'**(γ', var))
case $\mathsf{S}\gamma'$ **do**
(**let** var $\leftarrow$ (S curvar) **in pgm'**(γ', var))

Algorithm 3: Algorithm to construct functions corresponding to grammars in CG'.

Lemma 2.7 *Consider a grammar G in CG' with start symbol* S*. The grammar has a canonical derivation* $\mathsf{S} \stackrel{*}{\Rightarrow} \alpha$ *if and only if* $\mathsf{prop}((\mathsf{S}\ v), \{\epsilon\}, \mathtt{x}, \delta)$*, where* $\alpha \in \delta$*,* x *is the formal parameter of* S *and v is an arbitrary value of appropriate type.*

$\mathsf{S} \rightarrow L\mathsf{S}_1\mathbf{1}R$	$\mathsf{S}_1 \rightarrow \mathbf{0}\mathsf{S}_2\mathbf{0} \mid \mathbf{1}\mathsf{S}_3\mathbf{0}$

(**define** (S x)
  (**let** a ← (L x) **in**
  (**let** b ← (S_1 a) **in**
  (**let** c ← (**cdr** b) **in**
  (**let** d ← (R c) **in**
  (**return** d))))))
	(**define** (S_1 x)
	  (**if 0**
	  (**let** a ← (**car** x) **in**
	  (**let** b ← (S_2 a) **in**
	  (**let** c ← (**car** b) **in**
	  (**return** c))))
	  (**let** a ← (**cdr** x) **in**
	  (**let** b ← (S_3 a) **in**
	  (**let** c ← (**car** b) **in**
	  (**return** c))))))

Figure 2.10: Sample grammar rules and the corresponding programs generated by pgm.

Instead of proving Lemma 2.7, we prove the following generalization.

Lemma 2.8 *Consider a grammar G in CG'. For any non-terminal symbol T and a string $\alpha \in (\mathbf{0}+\mathbf{1}+\bar{\mathbf{0}}+\bar{\mathbf{1}})^*$, the grammar, $\mathsf{T}\alpha \stackrel{*}{\Rightarrow} \beta$ in a canonical derivation, if and only if $\mathsf{prop}((\mathsf{T}\ v), \{\alpha\}, \mathtt{x}, \delta)$, where $\beta \in \delta$, $\mathtt{x}$ is the formal parameter of T and v is any value of an appropriate type.*

PROOF. Assume without loss of generality that β is the string obtained without applying any of the rules in unrestricted, so that β is $\alpha'\alpha$ for some α' derivable from T.

First consider only the if part. The proof is by induction on the number of steps in the derivation. As the base case, consider a 1-step derivation of $\mathsf{T} \stackrel{*}{\Rightarrow} \alpha'\alpha$. This means that α' is ϵ and the production used is $\mathsf{T} \rightarrow \epsilon$. The program fragment corresponding to this choice of production for T simply returns the formal parameter $\mathtt{x}$ of T. It follows from the rules of DGS that $\mathsf{prop}((\mathsf{T}\ v), \{\alpha\}, \mathtt{x}, \{\alpha\})$.

Now consider the case where $\mathsf{T} \stackrel{*}{\Rightarrow} \alpha'$ in n steps. For the first step of the derivation, we have to do a case analysis on all productions with a non-ϵ RHS. Let us consider only one of them, say $\mathsf{T} \rightarrow \mathbf{0}\mathsf{T}'\mathbf{1}$, we can reason about other productions similarly. Further, we assume that $\mathsf{T}' \stackrel{*}{\rightarrow} \alpha''$ and since this derivation takes $n-1$ steps, we assume as induction hypothesis $\mathsf{prop}((\mathsf{T}'\ v), \{\mathbf{1}\alpha\}, \mathtt{y}, \delta')$ where $\alpha''\mathbf{1}\alpha \in \delta'$, $\mathtt{y}$ is the formal parameter of T' and v' is some value.

From the program corresponding to T, it is clear that (a) the CDR rule will prefix a $\mathbf{1}$ to the demand $\{\alpha\}$ on the return value of T which then becomes the demand on the

function call (T' a), (b) from the induction hypothesis and because of **id** rule used while binding the actual to the formal parameter y during a function call, the actual argument a appears on the DGS trace with a demand δ' containing $\alpha''\mathbf{1}\alpha$ and (c) the CAR rule adds a **0** to this demand giving the demand on the formal argument of T that contains $\mathbf{0}\alpha''\mathbf{1}\alpha$, or $\alpha'\alpha$.

Now consider the if part. We show by an induction on the depth of calls that if $\mathsf{prop}((\mathsf{T}\ v), \{\alpha\}, \mathtt{x}, \delta')$ such that $\alpha'\alpha \in \delta'$, then $(\mathsf{T}\ v) \xrightarrow{*} \alpha'\alpha$. The proof is by induction on the call depth of $(\mathsf{T}\ v)$. As base case, assume that the call depth of $(\mathsf{T}\ v)$ is 0, i.e. $(\mathsf{T}\ v)$ results in no further calls. Then the program fragment executed in T immediately returns the formal parameter, say x of T. From the rules of DGS, $\mathsf{prop}((\mathsf{T}\ v), \{\alpha\}, \mathtt{x}, \{\alpha\})$, or in other words α' is ϵ. Also, it follows from the description of **pgm** that corresponding to the program fragment above, T has a production that goes to ϵ, and therefore $\mathsf{T}\alpha \stackrel{*}{\Rightarrow} \alpha$.

For the inductive case, we once again consider a single illustrative function definition:

```
(define (T x)
   (if 0
     (let a ← (car x) in
       (let b ← (T' a) in
         (let c ← (cdr b) in
           (return c))))
     ...
```

As induction hypothesis, assume that $\mathsf{prop}((\mathsf{T}'\ \mathtt{a}), \{\mathbf{1}\alpha\}, \mathtt{y}, \delta')$ such that $\alpha'' \in \delta'$ implies $\mathsf{T}'\mathbf{1}\alpha \stackrel{*}{\Rightarrow} \alpha''\mathbf{1}\alpha$, where y is the assumed formal parameter of T'. Further, from the CDR rule of *DGS* and the induction hypothesis, we have $\mathsf{prop}((\mathsf{T}'\ \mathtt{a}), \{\mathbf{1}\alpha\}, \mathtt{y}, \delta')$ such that $\alpha''\mathbf{1}\alpha \in \delta'$ and the fact that a is bound to (id y) gives $\mathsf{prop}((\mathsf{T}\ v), \{\alpha\}, \mathtt{a}, \delta')$ Finally, by the CAR rule, $\mathsf{prop}((\mathsf{T}\ v), \{\alpha\}, \mathtt{x}, \delta)$ such that $\mathbf{0}\alpha''\mathbf{1}\alpha \in \delta$, so that α' is $\mathbf{0}\alpha''\mathbf{1}$. Now, **pgm** dictates that the program fragment must have been generated from the production $\mathsf{T} \rightarrow \mathbf{0}\mathsf{T}'\mathbf{1}$, and thus $\mathsf{T}\alpha \stackrel{*}{\Rightarrow} \mathbf{0}\alpha''\mathbf{1}\alpha$, or $\mathsf{T}\alpha \stackrel{*}{\Rightarrow} \alpha'\alpha$ ■

Theorem 2.9 *The dependence analysis problem is undecidable.*

PROOF. Assume to the contrary that there is an algorithm $\mathbb{A}$ for dependence analysis. Then given a grammar $G \in CG'$, we construct a program P that consists of a main program defined as (**define** (**main**) (**let** a ← (S v) **in** (**return** a))) where v is a value of an appropriate type. By Lemma 2.7, the ϵ-recognition problem of G, translates to the predicate $\exists\delta'\mathsf{prop}(P, \{\epsilon\}, \mathtt{x}, \delta')$ such that $\epsilon \in \delta'$ and x is the formal parameter of S. This can be answered by using $\mathbb{A}$ as $\mathbb{A}(P, \{\epsilon\}, \mathtt{x}, \epsilon)$. However, since the ϵ-recognition problem

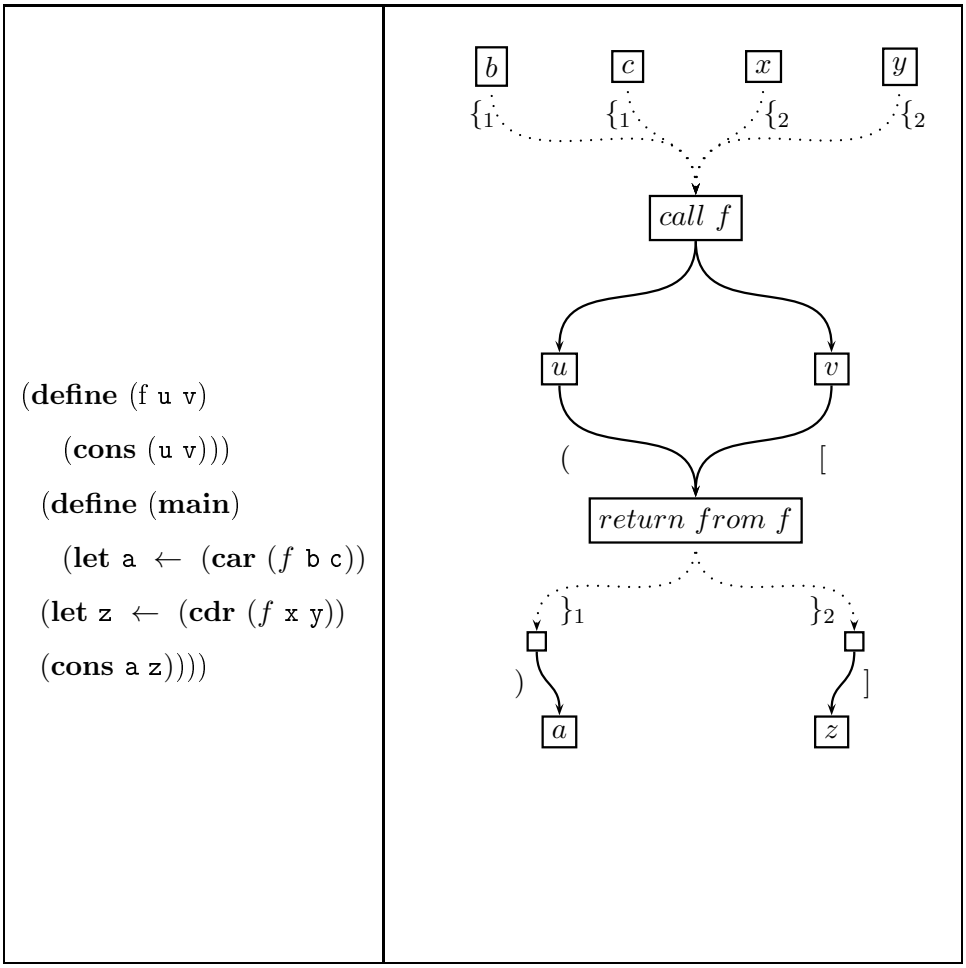


Figure 2.11: (a) Example scheme program (b) Labelled dependence graph corresponding to the program in (a). Dotted edges indicate interprocedural dependence, $\{_i$ $\}_i$ pair indicate matching call-return, (indicates putting a value in car part, [indicates putting value in cdr part,) indicates a car selection and] a cdr selection. A valid dependence will have a path in which all parenthesis match ({ and (([ )) can be interleaved).

of G has been shown to be undecidable, no such $\mathbb{A}$ can exist. Hence, the dependence analysis problem is undecidable. ■

2.4.3 Related work

The problem of devising a context-sensitive precise dependence analysis that can handle structure-transmitted dependence has been shown to be undecidable by Reps [78]. Unlike our reduction, the undecidability is shown by reducing a variant of the Post's Correspondence Problem (PCP) called parenthesis PCP or P-PCP.

Reps models dependence analysis as a graph reachability problem on a directed

graph, where the nodes of the graph represent program variables and the edges represent data dependence. A variable `u` is dependent on variable `v` if and only if there exists a path between the nodes corresponding to `u` and `v` satisfying some property. Figure 2.11 shows an example program and the corresponding graph. The function f takes two arguments `u` and `v` and returns a **cons** cell with `u` as the car-part and `v` as cdr-part. The main expression has two calls to f, the car-part of the result of the call (f `b c`) is assigned to `a` and the cdr-part of the result of the call (f `y z`) is assigned to `x`. A dependence analysis would answer questions such as: Is `a` dependent on `b`? What are the variables on which `a` may be dependent? Ignoring the property that needs to be satisfied, and, as a consequence, the labels on the edges for the time being, we will try to answer these questions by doing a reachability check on the graph shown in Figure 2.11(b). Variable `a` depends on `b` as there is a path in the graph between the nodes corresponding to `a` and `b`. Notice, however, that since there exists a path between `c` and `a`, we conclude that `a` is dependent on `c` and for similar reasons, also `a` is dependent on `y`. However, it is clear from the program that these dependences are spurious as `a` is not dependent on `c` when one considers structure transmitted dependence (captured by rules of the kind (**car** (**cons** `x y`)) = `x`), and `a` is not dependent on `y` when context-sensitivity (dependences are propagated only along matched call-return paths) is considered. These spurious dependences are generated because simple reachability cannot capture structure-transmitted dependences or context-sensitivity.

Reps uses the fact that program-analysis problems can be tackled by modelling them as CFL-reachability [77, 104] on labelled graphs. In a labelled graph, a node t is CFL-reachable from s if the string obtained from concatenating the labels on the path belongs to a given context-free language defining the property that such a path should have. In particular, context-sensitivity can be modelled by adding labels $\{_i$ $\}_i$ to call-return edges and defining a context-free grammar, say G_1 that accepts only those paths that have matched $\{_i$ $\}_i$. In the graph in Figure 2.11, considering the CFL-reachability using G_1, it is clear that the spurious path connecting `a` with `y` will be invalid. Similarly, we can add (to represent a value being passed as the first argument of **cons** [to represent the value being passed as the second argument of **cons**,) to represent **car** selection and] to represent a **cdr** selection and a different context-free grammar G_2 accepting only matched-parenthesis paths. Again, in this case, considering only paths along which the parenthesis (both ( ) and [ ]) are correctly matched, the fact that `a` is dependent on `c`

can be ruled out. However, notice that just considering context-sensitivity or structure-transmitted dependence alone is not sufficient to rule out all spurious paths and a fully precise dependence analysis should consider both. In terms of the graph, only paths along which both sets of parenthesis match ({ } for context-sensitivity and ( ) [ ] for structure-transmitted dependences) should be considered valid i.e. only paths in the language $G_1 \cap G_2$. The reader can verify that the path from `b` to `a` is valid and paths from `c` to `a` and `y` to `a` are invalid. Therefore, dependence analysis problem reduces to finding an algorithm that finds all and only fully matched paths of the type described above in the data dependence graph of a program.

Reps shows that this question is undecidable by reducing a variant of the Post's Correspondence Problem (PCP) to finding parenthesis matched paths. The PCP problem is defined as follows: Given 2 lists of k strings X and Y over the language $(\mathbf{0} + \mathbf{1})^+$, an instance of PCP has a solution if there exists a non-empty sequence of indices $i_1, i_2, \ldots i_m$ where $1 \leq k \leq m$ and $x_{i_1} x_{i_2} \ldots x_{i_m} = y_{i_1} y_{i_2} \ldots y_{i_m}$. The following instance of the PCP problem from [78] where, $X = \{0101, 101, 111\}$ and $Y = \{01, 011, 0111101\}$ has the solution $1, 2, 3, 1$ because,

$$x_1 x_2 x_3 x_1 = 01011011110101 = y_1 y_2 y_3 y_1$$

Reps introduces a variant of PCP called Parenthesis-PCP (P-PCP) and shows how to construct an instance of P-PCP given an instance of PCP. Given an instance of PCP with $X = x_1 x_2 x_3 \ldots x_k$ and $Y = y_1 y_2 y_3 \ldots y_k$ we construct the instance of P-PCP as,
$\overline{X} = \overline{x_1}\ \overline{x_2}\ \ldots\ \overline{x_k}$
$\overline{Y}^R = \overline{y_1}^R\ \overline{y_2}^R\ \ldots\ \overline{y_k}^R$
where ,

1. $\overline{x_i}$ are constructed by replacing $\mathbf{0}$ in x_i by (and $\mathbf{1}$ by [.
2. $\overline{y_i^R}$ are constructed by replacing $\mathbf{0}$ in y_i by) and $\mathbf{1}$ by] and then reversing the string. is reversed.

An instance of P-PCP is said to have a solution if it has a non-empty sequence

$$\overline{x_{i_1}}\ \overline{x_{i_2}}\ \ldots\ \overline{x_{i_m}}\ \#\ \overline{y_{i_m}}^R\ \ldots\ \overline{y_{i_2}}^R\ \overline{y_{i_1}}^R$$

where for all $1 \leq m$, we have $1 \leq i_j \leq k$ and the parenthesis are matched. The corresponding sets $\overline{X}$ and $\overline{Y}^R$ for the earlier instance are, $\overline{X} = \{([([, [([, [[[\}$ and $\overline{Y^R} = \{]),]]),])]]]]\}$.

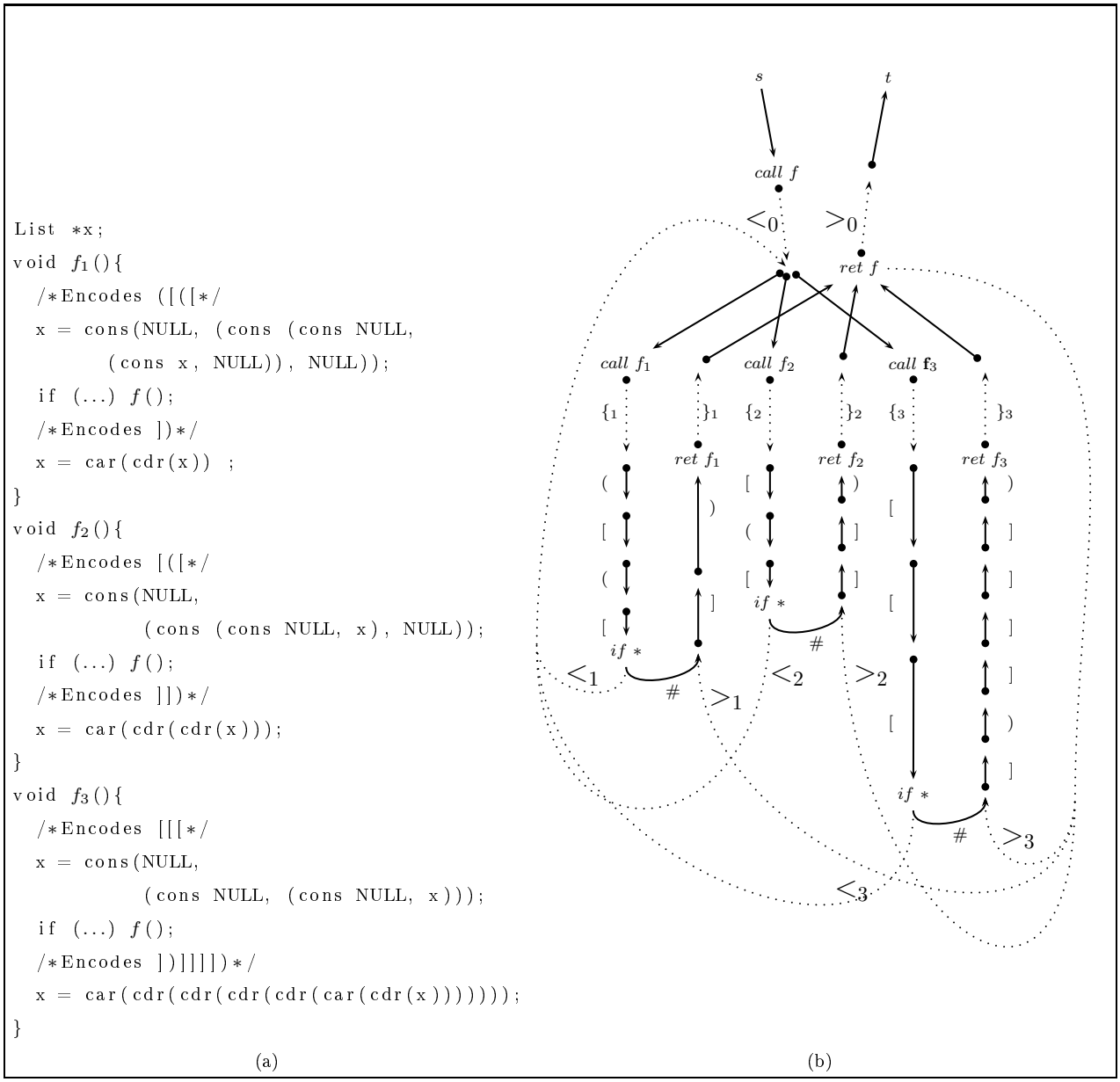


Figure 2.12: (a)Program corresponding to the P-PCP instance under discussion (b) Dependence graph for the program in (a).

It can be verified that the sequence 1,2,3,1 is a solution to this instance of P-PCP also. It is clear from the construction that if an instance of P-PCP has a solution then the corresponding instance of PCP also has a solution. This shows that P-PCP is also undecidable. Reps shows that given an instance of P-PCP, one can construct a program such that if and only if there exists a parenthesis matched path in the dependence graph then the corresponding P-PCP problem has a solution.

The program fragment that is equivalent to the instance of the P-PCP problem is

```
void f(){
  if (...) f1();
  elseif (...) f2()
  ...
  else fk();
}
```

```
void main(){
  s: x = atom(A);
  f();
  t: /* can x have the value A here?*/
}
```

Figure 2.13: The structure of **main** and the common function f.

shown in Figure 2.12. Each function f_i in the program encodes the $\overline{x_i}$ as a sequence of constructor operations and the corresponding $\overline{y_i}^R$ as a sequence of selectors. For example, the sequence for $\overline{x_1}$, ([([is encoded as $cons(NULL, (cons(consNULL, (consx, NULL)), NULL))$ and $\overline{y_1}^R$,]) as $car(cdr(x))$. A non-deterministic condition ensures that each f_i can call any other f_j (including itself) any number of times. Function call and returns are matched using the symbols $\{_i$ $\}_i$ ($<_i$ $>_i$). Functions f and **main** are shown in Figure 2.13.

From the construction it is clear that any f_i can be called from f any number of times and in any order, capturing the fact that any $\overline{x_i}$ can be used any number of times and in any order. Once the functions start returning from f only the code corresponding to $\overline{y_i}^R$s will be executed. A context-sensitive dependence analysis capable of correctly modelling structure transmitted dependences should be able to identify that variable `x` may have the value atom(A) at program point t (because the corresponding instance of P-PCP has the solution 1,2,3,1). It can be verified that the path, $<_0$ $\{_1$ ([([$<_1$ $\{_2$ [([$<_2$ $\{_3$ [[[$<_3$ $\{_1$ ([([ ]) $\}_1$ $>_3$])]]]]) $\}_3$ $>_2$]]) $\}_2$ $>_1$]) $\}_1$ $>_0$ corresponding to the sequence 1,2,3,1 is indeed well matched. The undecidability of context-sensitive structure transmitted data dependence analysis follows from the fact that if we had an algorithm that could check if such a path exists in the dependence graph then we could solve the P-PCP problem which is already known to be undecidable.

The undecidability proof presented in this thesis differs from Reps' in two respects, 1) While Reps uses PCP to show the undecidability we use the Turing machine halting problem 2) our proof is tightly coupled with the operational semantics that we have defined to give a formal definition of dependence analysis. The fact that it is closer to the operational semantics allows us to define an analysis which computes safe over-approximation

of dependences which we discuss in the following chapter. Most approximate analyses drop the requirement of either context-sensitivity [79] or structure transmitted data dependence [93] to become decidable. As we shall see later, our approximate dependence analysis is modelled as the emptiness question on intersection of 2 CFGs. However, it approximates the requirement of context-sensitivity by a regular grammar instead of a CFG. While the emptiness question of intersection of 2 CFGs is known to be undecidable, the emptiness question of the intersection of a CFG with a regular grammar is decidable [64]. We model structure transmitted dependence precisely using a CFG but over approximate the CFG corresponding to context-sensitivity by a regular grammar and use the intersection to compute approximate dependences. An analysis which is only context-sensitive or which only models structure transmitted dependences cannot eliminate spurious paths in the example in Figure 2.11, however the analysis that we will be presenting in the next chapter rules out all spurious dependences for the example.

Chapter 3

An approximate dependence analysis and its proof of correctness

In the previous chapter, we formulated the problem of dependence analysis and showed that computing precise dependence information is undecidable. In this chapter, we describe an analysis to compute an over-approximation of dependences for first-order functional programs. Interestingly, our formulation of the problem leads naturally to the approximate analysis. Our analysis is driven with a demand supplied by the user on a designated expression, which, for the applications considered in this thesis, namely liveness-based garbage collection and slicing, is a call to the function **main**. The user supplied demand will be denoted $\sigma_{\mathbf{main}}$. The result of the analysis is a (non-context-free) grammar corresponding to each expression in the program. These grammars effectively describe the parts of the expression on which the result of **main** is dependent. Answering queries related to dependence questions amounts to finding out membership of access paths in the generated grammars. Since we have already shown that precise dependence analysis is undecidable, the undecidability manifests in the membership question also turning out to be undecidable. We get around this undecidability by settling for an approximate answer to the membership problem. Finally, we prove the correctness of our formulation of dependence analysis using DGS.

3.1 An approximate dependence analysis

We now describe an analysis to compute dependences in functional programs. As mentioned earlier, this analysis should be interprocedural, since the central construct in a functional program is a function call. In the interest of precision, the handling of function calls should be context sensitive, and for reasons of efficiency, a function body should not be analyzed more than once. Finally, for an accurate modelling of the state of the heap, the interaction between constructors and selectors should be modeled as part of the analysis, in other words we should model structure-transmitted dependence in the model of Reps [78].

The analysis that we consider addresses scalability and precision concerns by computing context independent summaries of the effect of functions and then using these summaries at call sites to mimic the effect of the function call on its arguments. These summaries act as demand transformers which transform the demand on the function call to demand on the arguments of the call. Thus, if the demands on different calls to the same function are different, the demands propagated to the arguments will also be different. This effectively captures context sensitivity.

Figure 3.1 describes our analysis. First notice that a null demand (denoted by $\emptyset$) on any expression[1] results in a null demand on the constituents of the expression. This captures the fact that when no part of the value of the expression is required, none of its constituents need to be computed. This means that the evaluation of the language is a generalization of lazy evaluation—the extent of evaluation of an application or expression is determined by the demand on it.

The function $\mathcal{A}$, takes an application s and a demand σ and returns a demand environment that maps the demand on each argument of s (represented by its program point) due to the application. The third parameter to $\mathcal{A}$, denoted $\mathbb{DS}$, represents context-independent summaries of the functions in the program and is used to analyze function calls. This will be explained shortly. A *demand environment* is a mapping from program points to a demand, expressed as $\{\pi_1 \mapsto \{\epsilon, \mathbf{1}, \mathbf{11}\}, \pi_2 \mapsto \{\epsilon\}, \pi_3 \mapsto \{\}\}$. In this notation, $\pi_2 \mapsto \{\epsilon\}$ indicates that the demand on the expression at π_2 is the root of the value of the expression. Similarly $\pi_3 \mapsto \{\}$ indicates an empty demand on the expression at π_3. We

[1] In general, we shall call elements of both *Expr* and *App* expressions, distinguishing them only when required by the context

$$\boxed{\mathcal{A} :: (\mathsf{App}, \mathsf{Demand}, \mathsf{FuncSummaries}) \to \mathsf{DemandEnvironment}}$$

$$\begin{aligned}
\mathcal{A}(\pi{:}\kappa, \sigma, \mathbb{DS}) &= \{\pi \mapsto \sigma\}, \text{ for constants including } \mathbf{nil} \\
\mathcal{A}(\pi{:}(\mathbf{null?}\ \pi_1{:}x), \sigma, \mathbb{DS}) &= \{\pi_1 \mapsto \text{if } \sigma \neq \emptyset \text{ then } \{\epsilon\} \text{ else } \emptyset\} \\
\mathcal{A}(\pi{:}(+\ \pi_1{:}x\ \pi_2{:}y), \sigma, \mathbb{DS}) &= \{\pi_1 \mapsto \text{if } \sigma \neq \emptyset \text{ then } \{\epsilon\} \text{ else } \emptyset, \\
&\qquad \pi_2 \mapsto \text{if } \sigma \neq \emptyset \text{ then } \{\epsilon\} \text{ else } \emptyset\} \\
\mathcal{A}(\pi{:}\ (\mathbf{car}\ \pi_1{:}\ x), \sigma, \mathbb{DS}) &= \{\pi_1 \mapsto \text{if } \sigma \neq \emptyset \text{ then } \mathbf{0}\sigma \text{ else } \emptyset\} \\
\mathcal{A}(\pi{:}\ (\mathbf{cdr}\ \pi_1{:}\ x), \sigma, \mathbb{DS}) &= \{\pi_1 \mapsto \text{if } \sigma \neq \emptyset \text{ then } \mathbf{1}\sigma \text{ else } \emptyset\} \\
\mathcal{A}(\pi{:}(\mathbf{cons}\ \pi_1{:}x\ \pi_2{:}y), \sigma, \mathbb{DS}) &= \{\pi_1 \mapsto \{\alpha \mid \mathbf{0}\alpha \in \sigma\}, \pi_2 \mapsto \{\beta \mid \mathbf{1}\beta \in \sigma\}\} \\
\mathcal{A}(\pi{:}(f\ \pi_1{:}y_1\ \cdots\ \pi_n{:}y_n), \sigma, \mathbb{DS}) &= \textstyle\bigcup_{i=1}^{n} \{\pi_i \mapsto \mathbb{DS}_f^i(\sigma)\}
\end{aligned}$$

$$\boxed{\mathcal{D} :: (\mathsf{Exp}, \mathsf{Demand}, \mathsf{FuncSummaries}) \to \mathsf{DemandEnvironment}}$$

$$\begin{aligned}
\mathcal{D}(\pi{:}(\mathbf{return}\ \pi_1{:}x), \sigma, \mathbb{DS}) &= \{\pi_1 \mapsto \sigma,\ \pi \mapsto \sigma\} \\
\mathcal{D}(\pi{:}(\mathbf{if}\ \pi_1{:}x\ e_1\ e_2), \sigma, \mathbb{DS}) &= \mathcal{D}(e_1, \sigma, \mathbb{DS}) \cup \mathcal{D}(e_2, \sigma, \mathbb{DS}) \cup \\
&\qquad \{\pi_1 \mapsto \text{if } \sigma \neq \emptyset \text{ then } \{\epsilon\} \text{ else } \emptyset,\ \pi \mapsto \sigma\} \\
\mathcal{D}(\pi{:}(\mathbf{let}\ x\ \leftarrow\ \pi_1{:}s\ \mathbf{in}\ e), \sigma, \mathbb{DS}) &= \mathcal{A}(s, \sigma', \mathbb{DS})\ \cup\ \{\pi \mapsto \sigma, \pi_1 \mapsto \sigma'\} \\
&\qquad \text{where } \Pi \text{ is the set of program points} \\
&\qquad \text{representing all occurrences of } x \text{ in } e \\
&\qquad \mathrm{DE} = \mathcal{D}(e, \sigma, \mathbb{DS}),\ \text{and } \sigma' = \cup_{\pi' \in \Pi} \mathrm{DE}(\pi'),
\end{aligned}$$

$$\boxed{\mathbb{DS} \in \mathsf{FuncSummaries} :: \mathsf{Funcname} \to (\mathsf{Demand} \to (\mathsf{Demand}_1, \ldots, \mathsf{Demand}_n))}$$

$$\frac{\forall f, \forall i, \forall \sigma :\ \ \mathcal{D}(e_f, \sigma, \mathbb{DS}) = \mathrm{DE}, \mathbb{DS}_f^i = \bigcup_{\pi \in \Pi}\ \mathrm{DE}(\pi)}{df_1 \ldots df_k\ \vdash^l\ \mathbb{DS}}\ (\textsc{function-summaries})$$

$(\mathbf{define}\ (f\ z_1\ \ldots\ z_n)\ \ e_f)$ is one of $df_1\ \ldots df_k$, $1 \leq i \leq n$,

and Π represents all occurrences of z_i in e_f

Figure 3.1: Demand equations and judgment rule

use DE to range over demand environments. The demand on the expression at a program point π is denoted as $\mathsf{DE}(\pi)$, but can also be written as DE_π.

Now the $\mathcal{A}$ rules: A demand of σ on the application $(\mathbf{car}\ x)$, is transformed to the demand $\mathbf{0}\sigma$ on the argument x. This is illustrated in Figure 3.2(a). To compute σ of $(\mathbf{car}\ x)$, we have to start with the root of x, dereference using the **car** field and then compute σ of the tree thus obtained, resulting in the path $\mathbf{0}\sigma$. The rule for $(\mathbf{cdr}\ x)$ is

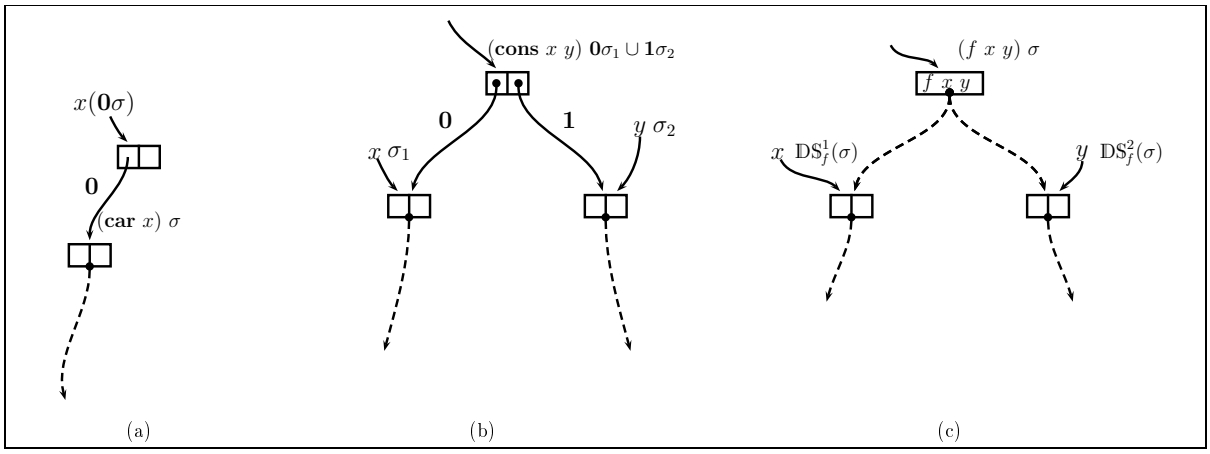


Figure 3.2: Illustration of application rules (a) A demand of σ on (**car** x) resulting in a demand of $\mathbf{0}\sigma$ on x (b) **cons** rule (c) Function application.

similar. In an opposite sense, illustrated in Figure 3.2(b), the demand of $\mathbf{0}\sigma_1$ on (**cons** x y) is transformed to the demand σ_1 on x and a $\emptyset$ demand on y , and a demand of $\mathbf{1}\sigma_2$ on (**cons** x y) is transformed into a demand of σ_2 on y and $\emptyset$ demand on x. Since (**null**? x) only requires the root of x to examine the constructor, a non-null demand on (**null**? x) translates to the demand ϵ on x. A similar reasoning also explains the rule for $(+\ x\ y)$. Since, both x and y evaluate to integers in a well typed program, a non-null demand on $(+\ x\ y)$ translates to the demand ϵ on both x and y.

Just as $\mathcal{A}$ defines how a primitive like **car** maps a demand on itself to demands on its arguments, we would like to derive a similar transformation for user-defined functions. Since user-defined functions are, in general, mutually dependent, we define this transformation simultaneously for all user-defined functions. This is given by the inference rule DEMAND-SUMMARY and results in a set of functions $\mathbb{DS}_f^i$, defining how a demand σ on a call to f is propagated to its ith parameter. The rule for function calls uses $\mathbb{DS}$ to propagate demands to the arguments of a specific call. We look upon the functions for $\mathbb{DS}_f$ as a context-independent summary of f—context-independent because it is parameterized with respect to the demand that will be instantiated at the place where the function is called.

The rule FUNCTION-SUMMARIES specifies the fixed-point property to be satisfied by $\mathbb{DS}$, namely, the demand transformation assumed for each function in the program should be the same as the demand transformation calculated from the body of the function. The reader will notice the similarity between this rule and the rule for recursive *let*s in

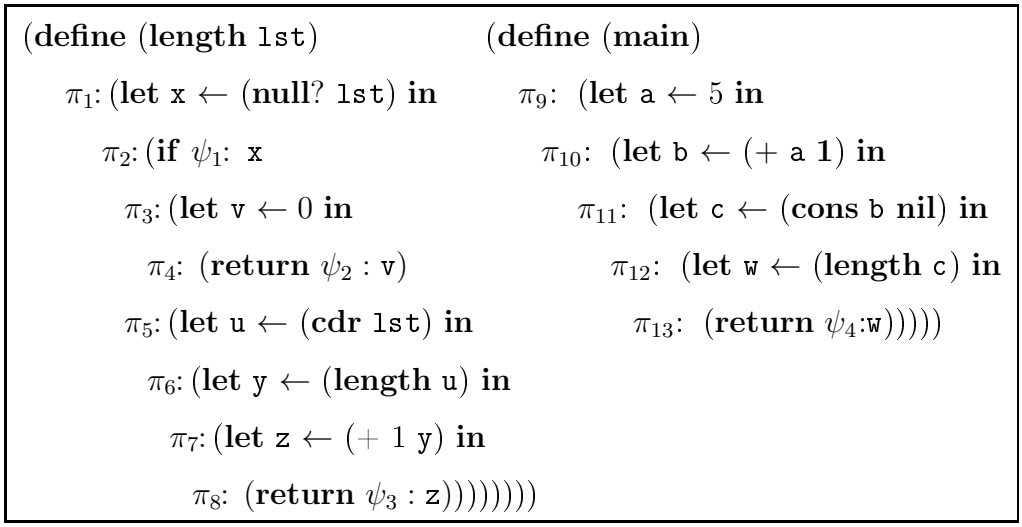


Figure 3.3: An example program

the Hindley-Milner system of type inference [34, 59, 76]. An operational interpretation of the rule to find $\mathbb{DS}^i_f(\sigma)$ proceeds by analyzing e_f, the body of f, with respect to a symbolic demand σ. Then $\mathbb{DS}^i_f(\sigma)$ is the union of the demands on all occurrences of the ith argument in e_f. A call to a function, say g, in e_f is analyzed using the summary $\mathbb{DS}_g$. In general, this results in a recursive description of $\mathbb{DS}^i_f(\sigma)$. We explain in Section 3.2 how to convert this to a closed form.

We next describe the function $\mathcal{D}$ that propagates demands across expressions. Consider the $\mathcal{D}$-rules for **let**, **if**, and **return**. Since the value of (**return** x) is the value of x, a demand σ on (**return** x) gives a demand of σ. The demand of the expression (**if** x e_1 e_2) is a union of the demands of e_1 and e_2. In addition, since the condition x is also evaluated, the demand $\{\epsilon\}$ is created and added to the union. Note that, the condition x needs to be evaluated only if the result of the **if** expression is required, i.e. demand on σ is not $\emptyset$. Hence, the demand $\{\epsilon\}$ is added only if the demand on **if** is not $\emptyset$. Finally, since the value of (**let** $x \leftarrow s$ **in** e) is the value of its body e, the rule for **let** first uses σ to calculate the demand environment DE of e. The demand on s is the union of the demands on all occurrences of x in e. Notice that the demand environment for each expression e also includes the demand on e itself apart from its subexpressions.

3.1.1 An Example

For the rest of the chapter, we consider the program in Figure 3.3 as our running example. The program takes a list as input and computes its length. Consider $\mathbb{DS}^1_{\mathbf{length}}$, the function

that propagates the demand on a call to **length** to its first (in fact, only) argument. An operational interpretation of the rule FUNCTION-SUMMARIES rule requires us to do a dependence analysis of the body of **length** with a symbolic demand σ, union the resulting demands on all occurrences of the argument `lst` in the body, and equate it to $\mathbb{DS}^1_{\mathbf{length}}$. Assume for the sake of simplicity that σ is not $\emptyset$. Firstly notice that, according to the rules of **let** and **if**, the demand on `z` is also σ. This is propagated to `y` through (+ 1 `y`), which in turn is propagated to `u` through (**length** `u`), and finally to `lst` through (**cdr** `lst`). The reader can verify that the resulting demand on this occurrence of `lst` is $\mathbf{1}\mathbb{DS}^1_{\mathbf{length}}(\epsilon)$. On the other hand the demand on `x` is also ϵ (the **if** rule), and this is propagated to the `lst` in (**null**? `lst`) resulting in a demand of ϵ for this occurrence of `lst`. Thus:

$$\mathbb{DS}^1_{\mathbf{length}} = \epsilon \cup \mathbf{1}\mathbb{DS}^1_{\mathbf{length}}(\epsilon)$$

Notice that this equation is recursive in $\mathbb{DS}^1_{\mathbf{length}}$, and in order to be able to use it to compute dependences, we have to bring it to a closed form.

3.2 Computing dependences

The analysis in Section 3.1 is precise and context-sensitive, describing the demands inside a function body in terms of a *symbolic demand* σ and the function summaries $\mathbb{DS}$. What we have not said so far is how the demand on the function body is to be determined. This is as follows:

1. The demand on the body of the main program, $e_{\mathbf{main}}$, is user supplied, and is denoted by $\sigma_{\mathbf{main}}$.
2. The demand on a function body e_f is the union of demand over all calls to f.

The function bodies are analyzed using the demands described above. The advantage of summarizing a function in a context-sensitive manner using a symbolic demand is that while analyzing a function body, it helps us to propagate a demand across several calls to a function without analyzing its body each time. Additionally, as we shall show in Chapter 5, it is the key to our incremental slicing method.

However, some of the rules of dependence analysis requires us to do operations that cannot be done on a symbolic demand. For example, the **cons** rule defined as $\{\alpha \mid \mathbf{0}\alpha \in \sigma\}$ clearly requires us to know strings belonging to σ that start with **0**.

Recall from Section 2.4.1, we were able to describe the set operations of the **cons** rule algebraically by introducing symbols $\bar{\mathbf{0}}$ and $\bar{\mathbf{1}}$. We replace the rule $\{\alpha \mid \mathbf{0}\alpha \in \sigma\}$ with the rule $\bar{\mathbf{0}}\sigma$ and $\{\alpha \mid \mathbf{1}\alpha \in \sigma\}$ with $\bar{\mathbf{1}}\sigma$. While $\mathbf{0}$ represents selection using the **car** selector, the symbol $\bar{\mathbf{0}}$ represents the use of a value as the first argument of **cons**. Thus, $\bar{\mathbf{0}}\mathbf{0}$ represents first putting a value in the **car** part of a **cons** cell and following it with a **car** selection, effectively cancelling out each other. We therefore add the rule $\bar{\mathbf{0}}\mathbf{0} \to \epsilon$ to capture this fact. Similarly, for the analysis to handle lazy semantics, the **if** rule should place an ϵ demand on its conditional expression only if the incoming demand in non-null. We introduce the symbol $\boldsymbol{\emptyset}_{\boldsymbol{\epsilon}}$[2] to capture this operation. $\boldsymbol{\emptyset}_{\boldsymbol{\epsilon}}$ represents the symbolic transformation of any non-null demand to ϵ and null demand to itself. The simplification function $\mathcal{S}$ defines and makes these transformations deterministic.

$$
\begin{aligned}
\mathcal{S}(\{\epsilon\}) &= \{\epsilon\} \\
\mathcal{S}(\mathbf{0}\sigma) &= \mathbf{0}\mathcal{S}(\sigma) \\
\mathcal{S}(\mathbf{1}\sigma) &= \mathbf{1}\mathcal{S}(\sigma) \\
\mathcal{S}(\bar{\mathbf{0}}\sigma) &= \{\alpha \mid \mathbf{0}\alpha \in \mathcal{S}(\sigma)\} \\
\mathcal{S}(\bar{\mathbf{1}}\sigma) &= \{\alpha \mid \mathbf{1}\alpha \in \mathcal{S}(\sigma)\} \\
\mathcal{S}(\boldsymbol{\emptyset}_{\boldsymbol{\epsilon}}\sigma) &= \begin{cases} \emptyset & \text{if } \mathcal{S}(\sigma) = \emptyset \\ \{\epsilon\} & \text{otherwise} \end{cases} \\
\mathcal{S}(\sigma_1 \cup \sigma_2) &= \mathcal{S}(\sigma_1) \cup \mathcal{S}(\sigma_2)
\end{aligned}
$$

Notice that $\bar{\mathbf{0}}$ strips the leading $\mathbf{0}$ from the string following it, as required by the rule for **cons**. Similarly, $\boldsymbol{\emptyset}_{\boldsymbol{\epsilon}}$ examines the string following it and replaces it by $\emptyset$ or $\{\epsilon\}$; this is required by several rules. The $\mathcal{A}$ rules for **cons** and **null**? in terms of the new symbols are:

$$
\begin{aligned}
&\mathcal{A}(\pi{:}(\mathbf{cons}\ \pi_1{:}x\ \pi_2{:}y), \sigma, \mathbb{D}\$) = \{\pi_1 \mapsto \bar{\mathbf{0}}\sigma, \pi_2 \mapsto \bar{\mathbf{1}}\sigma\} \\
&\mathcal{A}(\pi{:}(\mathbf{null}?\ \pi_1{:}x), \sigma, \mathbb{D}\$) = \{\pi_1 \mapsto \boldsymbol{\emptyset}_{\boldsymbol{\epsilon}}\sigma\}
\end{aligned}
$$

[2]Odd as it may seem, choice of the symbol $\boldsymbol{\emptyset}_{\boldsymbol{\epsilon}}$ is to check whether the demand following it is the null demand $\emptyset$.

and the $\mathcal{D}$ rule for **if** is:

$$\mathcal{D}(\pi{:}(\textbf{if } \pi_1{:}x\ e_1\ e_2), \sigma, \mathbb{D}\$) = \mathcal{D}(e_1, \sigma, \mathbb{D}\$) \cup \mathcal{D}(e_2, \sigma, \mathbb{D}\$) \cup \{\pi_1 \mapsto \text{if } \sigma \neq \emptyset \text{ then } \{\epsilon\} \text{ else } \emptyset, \pi \mapsto \sigma\}$$

The rules for + are also modified similarly. We keep applying the simplification rules starting from the right, the simplification process stops when no rules are applicable. If the final string does not have any bar-edge symbols the string belongs to the language generated otherwise it does not. The following examples show the process of simplification,

$$\{\mathbf{1}\emptyset_\epsilon\bar{\mathbf{0}}\bar{\mathbf{1}}\mathbf{10}\} \xrightarrow{\mathcal{S}} \mathbf{1}\mathcal{S}(\{\emptyset_\epsilon\bar{\mathbf{0}}\bar{\mathbf{1}}\mathbf{10}\}) \xrightarrow{\mathcal{S}} \mathbf{1}\emptyset_\epsilon\mathcal{S}(\{\bar{\mathbf{0}}\bar{\mathbf{1}}\mathbf{10}\}) \xrightarrow{\mathcal{S}} \mathbf{1}\emptyset_\epsilon\bar{\mathbf{0}}\mathcal{S}(\{\bar{\mathbf{1}}\mathbf{10}\})$$
$$\xrightarrow{\mathcal{S}} \mathbf{1}\emptyset_\epsilon\bar{\mathbf{0}}\bar{\mathbf{1}}\mathcal{S}(\{\mathbf{10}\}) \xrightarrow{\mathcal{S}} \mathbf{1}\emptyset_\epsilon\bar{\mathbf{0}}\bar{\mathbf{1}}\mathbf{1}\mathcal{S}(\{\mathbf{0}\}) \xrightarrow{\mathcal{S}} \mathbf{1}\emptyset_\epsilon\bar{\mathbf{0}}\bar{\mathbf{1}}\mathbf{10}\mathcal{S}(\{\epsilon\})$$
$$\xrightarrow{\mathcal{S}} \mathbf{1}\emptyset_\epsilon\bar{\mathbf{0}}\bar{\mathbf{1}}\mathbf{1}\mathcal{S}(\{\mathbf{0}\}) \xrightarrow{\mathcal{S}} \mathbf{1}\emptyset_\epsilon\bar{\mathbf{0}}\bar{\mathbf{1}}\mathcal{S}(\{\mathbf{10}\}) \xrightarrow{\mathcal{S}} \mathbf{1}\emptyset_\epsilon\bar{\mathbf{0}}\mathcal{S}(\{\mathbf{0}\}) \xrightarrow{\mathcal{S}} \mathbf{1}\emptyset_\epsilon\mathcal{S}(\{\epsilon\}) \xrightarrow{\mathcal{S}} \mathbf{1}$$

In this example, the final string contains no bar-edge symbols and therefore is a valid string.

$$\{\mathbf{0}\emptyset_\epsilon\mathbf{0}\bar{\mathbf{1}}\bar{\mathbf{1}}\mathbf{1}\bar{\mathbf{0}}\} \xrightarrow{\mathcal{S}} \mathbf{0}\mathcal{S}(\{\emptyset_\epsilon\mathbf{0}\bar{\mathbf{1}}\bar{\mathbf{1}}\mathbf{1}\bar{\mathbf{0}}\}) \xrightarrow{\mathcal{S}} \mathbf{0}\emptyset_\epsilon\mathcal{S}(\{\mathbf{0}\bar{\mathbf{1}}\bar{\mathbf{1}}\mathbf{1}\bar{\mathbf{0}}\}) \xrightarrow{\mathcal{S}} \mathbf{0}\emptyset_\epsilon\mathbf{0}\mathcal{S}(\{\bar{\mathbf{1}}\bar{\mathbf{1}}\mathbf{1}\bar{\mathbf{0}}\})$$
$$\xrightarrow{\mathcal{S}} \mathbf{0}\emptyset_\epsilon\mathbf{0}\bar{\mathbf{1}}\mathcal{S}(\{\bar{\mathbf{1}}\mathbf{1}\bar{\mathbf{0}}\}) \xrightarrow{\mathcal{S}} \mathbf{0}\emptyset_\epsilon\mathbf{0}\bar{\mathbf{1}}\bar{\mathbf{1}}\mathcal{S}(\{\mathbf{1}\bar{\mathbf{0}}\}) \xrightarrow{\mathcal{S}} \mathbf{0}\emptyset_\epsilon\mathbf{0}\bar{\mathbf{1}}\bar{\mathbf{1}}\mathbf{1}\mathcal{S}(\{\bar{\mathbf{0}}\}) \xrightarrow{\mathcal{S}} \mathbf{0}\emptyset_\epsilon\mathbf{0}\bar{\mathbf{1}}\bar{\mathbf{1}}\mathbf{1}\bar{\mathbf{0}}\mathcal{S}(\{\epsilon\})$$
$$\xrightarrow{\mathcal{S}} \mathbf{0}\emptyset_\epsilon\mathbf{0}\bar{\mathbf{1}}\bar{\mathbf{1}}\mathbf{1}\bar{\mathbf{0}}\emptyset \xrightarrow{\mathcal{S}} \mathbf{0}\emptyset_\epsilon\mathbf{0}\bar{\mathbf{1}}\bar{\mathbf{1}}\mathbf{1}\emptyset \xrightarrow{\mathcal{S}} \mathbf{0}\emptyset_\epsilon\mathbf{0}\bar{\mathbf{1}}\bar{\mathbf{1}}\emptyset \xrightarrow{\mathcal{S}} \mathbf{0}\emptyset_\epsilon\mathbf{0}\bar{\mathbf{1}}\emptyset \xrightarrow{\mathcal{S}} \mathbf{0}\emptyset_\epsilon\mathbf{0}\emptyset \xrightarrow{\mathcal{S}} \mathbf{0}\emptyset_\epsilon\emptyset \xrightarrow{\mathcal{S}} \mathbf{0}\emptyset \xrightarrow{\mathcal{S}} \emptyset$$

In the second example, $\mathcal{S}$ generates an $\emptyset$ when it encounters an ϵ following a $\bar{\mathbf{0}}$ symbol. Once $\emptyset$ is generated, the semantics of concatenation of strings ensures that the final result of $\mathcal{S}$ is an empty string, indicating that the string is not valid.

Now the demand summaries can be obtained symbolically with the new symbols as markers indicating the operations that should be performed on the string following it. When the final demand environments are obtained with $\sigma_{\mathbf{main}}$ acting a concrete demand for the main expression $e_{\mathbf{main}}$, the symbols $\bar{\mathbf{0}}$, $\bar{\mathbf{1}}$ and $\emptyset_\epsilon$ are eliminated using the simplification function $\mathcal{S}$. The original rules and the modified rules are related through the simplification function $\mathcal{S}$ as follows:

Proposition 3.1 *Assume that a demand computation based on the original set of rules gives the demand on the expression $\pi : e$ as σ (symbolically, $\mathsf{DE}(\pi) = \sigma$). Further, let $\mathsf{DE}(\pi) = \sigma'$ when the modified rules are used instead of $\mathcal{D}$. Then $\sigma = \mathcal{S}(\sigma')$.*

To see why the proposition is true, consider an analysis based on the modified rules in which σ appears in the context $\mathcal{A}((\mathbf{cons}\ x\ y), \sigma, \mathsf{DS})$. Let $\alpha \in \sigma$. The symbol $\bar{\mathbf{0}}(\bar{\mathbf{1}})$ merely marks a place in α where the original **cons** rule would have erased an immediately following $\mathbf{0}(\mathbf{1})$, or, in absence of such a symbol, would have dropped α itself. Since the application of the modified rules merely add symbols at the beginning of α, the markers and other symbols in α are propagated to other dependent parts of program in their same relative positions. Consequently, the erasure carried out at the end of the analysis with $\mathcal{S}$ gives the same result as obtained through the original rules. The proposition also holds for other modified rules for similar reasons.

3.2.1 Obtaining closed form for function summaries $\mathbb{DS}$

As mentioned earlier, and illustrated in the example in the last section, to obtain the context-independent summary of a function f with respect to its ith argument, $\mathbb{DS}^i_f$, we start with a symbolic demand σ and compute the demand environment for e_f, the body of f. From this we calculate the overall demand on the ith argument of f, say x. This is the union of demands of all occurrences of x in e_f. This demand on the ith argument is equated to $\mathbb{DS}^i_f(\sigma)$. Since the body may contain other calls, the dependence analysis within e_f makes use of $\mathbb{DS}$ in turn. Thus, on the whole, $\mathbb{DS}$ will be given by a set of equations, one for every argument of each function. For the running example, the equation shown below defines $\mathbb{DS}^1_{\mathbf{length}}(\sigma)$. $\mathsf{DE}(\pi_1)$ and $\mathsf{DE}(\pi_5)$ are the demands on the two occurrences of `lst` in the body of **length**.

$$\mathbb{DS}^1_{\mathbf{length}}(\sigma) = \mathsf{DE}(\pi_1) \cup \mathsf{DE}(\pi_5) = \boldsymbol{\emptyset}_{\boldsymbol{\epsilon}}\sigma \cup \mathbf{1}\mathbb{DS}^1_{\mathbf{length}}(\boldsymbol{\emptyset}_{\boldsymbol{\epsilon}}\sigma)$$

This looks different from the equation for $\mathbb{DS}^1_{\mathbf{length}}(\sigma)$ in Sections 1.4.1 and 3.1.1 because of two reasons: We no longer assume that σ is non-empty, and the equation is written using the modified rules of dependence analysis that make use of the symbols $\bar{\mathbf{0}}$, $\bar{\mathbf{1}}$ and $\boldsymbol{\emptyset}_{\boldsymbol{\epsilon}}$.

As noted in [79], the main difficulty in obtaining a convenient function summary is to find a closed-form for $\mathbb{DS}^1_{\mathbf{length}}(\sigma)$ instead of the recursive description. Our solution to the problem lies in the following observation: Since we know that the rules of dependence analysis always prefix σ with symbols, we can write $\mathbb{DS}^i_f(\sigma)$ as $\mathsf{DS}^i_f\,\sigma$ (DS^i_f concatenated with σ), where DS^i_f is a set of strings over the alphabet $\{\mathbf{0}, \mathbf{1}, \bar{\mathbf{0}}, \bar{\mathbf{1}}, \boldsymbol{\emptyset}_{\boldsymbol{\epsilon}}\}$, and represents

the effect of $\mathbb{DS}_f^i$ on σ. The modified equation after substituting the guessed form of $\mathbb{DS}^1_{\mathbf{length}}(\sigma)$ in the equation will be:

$$\mathsf{DS}^1_{\mathbf{length}}\sigma = \boldsymbol{\emptyset}_{\boldsymbol{\epsilon}}\sigma \cup \mathbf{1}\mathsf{DS}^1_{\mathbf{length}}\boldsymbol{\emptyset}_{\boldsymbol{\epsilon}}\sigma$$

Substituting the guessed form in the equation describing $\mathbb{DS}_f$, and factoring out σ, we get an equation for D_f^i that is independent of σ. Applied to $\mathbb{DS}_{\mathbf{length}}$, we get:

$$\mathbb{DS}^1_{\mathbf{length}}(\sigma) = \mathsf{DS}^1_{\mathbf{length}}\sigma, \text{ and}$$
$$\mathsf{DS}^1_{\mathbf{length}} = \boldsymbol{\emptyset}_{\boldsymbol{\epsilon}} \cup \mathbf{1}\mathsf{DS}^1_{\mathbf{length}}\boldsymbol{\emptyset}_{\boldsymbol{\epsilon}}$$

Any solution for DS_f^i yields a solution for $\mathbb{DS}_f$. Note that the equation can also be viewed as a CFG with $\{\mathbf{1}, \boldsymbol{\emptyset}_{\boldsymbol{\epsilon}}\}$ as terminal symbols and $\mathsf{DS}^1_{\mathbf{length}}$ as the sole non-terminal.

3.2.2 Computing the demand environment for function bodies

While the computation of function summary assumed a symbolic demand for each function, to compute the demand environment, we have to supply the concrete demand for each function. The concrete demand on a function denoted as σ_f is computed in a manner similar to 0-CFA [91], by taking the union of the demands at all call-sites of f. This acts as a safe over-approximation and keeps the analysis sound. The demand environment of a function body e_f is calculated using σ_f. If there is a call to g inside e_f, the demand summary $\mathbb{DS}_g$ is used to propagate the demand across the call. Continuing with our example, we assume that the concrete demand on the body of **length** to be denoted by $\sigma_{\mathbf{length}}$ and the demand on $e_{\mathbf{main}}$ to be $\sigma_{\mathbf{main}}$. Since **length** has calls from **main** with a demand $\sigma_{\mathbf{main}}$ and a recursive call at π_6 with a demand $\boldsymbol{\emptyset}_{\boldsymbol{\epsilon}}\sigma_{\mathbf{length}}$. Thus:

$$\sigma_{\mathbf{length}} \;=\; \sigma_{\mathbf{main}} \;\cup\; \boldsymbol{\emptyset}_{\boldsymbol{\epsilon}}\sigma_{\mathbf{length}}$$

We calculate the demands on all expressions arguments of **length** in terms of $\sigma_{\mathbf{length}}$. Thus the demand on $\mathtt{u}$ at π_6, denoted by D_{π_6}, is $\mathsf{DS}^1_{\mathbf{length}}\boldsymbol{\emptyset}_{\boldsymbol{\epsilon}}\sigma_{\mathbf{length}}$.

At the end of this step, we shall have (i) A set of equations defining the demand summaries $\mathbb{DS}_f^i$ for each argument of each function, (ii) Equations specifying the demand D_π at each program point π, and (iii) an equation for each concrete demand σ_f on the body of each function f.

3.2.3 Converting equations to grammars:

Notice that the equations for $\mathsf{DS}^1_{\mathbf{length}}$ and $\sigma_{\mathbf{length}}$ are still recursive. However, these equations can also be viewed as a grammar with $\{\mathbf{0}, \mathbf{1}, \bar{\mathbf{1}}, \bar{\mathbf{0}}, \boldsymbol{\emptyset}_{\boldsymbol{\epsilon}}\}$ as terminal symbols and $\mathsf{DS}^1_{\mathbf{length}}$, D_{π_6} and $\sigma_{\mathbf{length}}$ as non-terminals. Thus finding the solution to the set of equations generated by the dependence analysis reduces to finding the language generated by the corresponding grammar. In fact the language generated by the grammar is the least solution of equations above. The least solution corresponds to the most precise dependence analysis. The equations can now be re-written as grammar rules:

$$\begin{aligned} \mathsf{D}_{\pi_6} &\rightarrow \mathsf{DS}^1_{\mathbf{length}} \boldsymbol{\emptyset}_{\boldsymbol{\epsilon}} \sigma_{\mathbf{length}} \\ \mathsf{DS}^1_{\mathbf{length}} &\rightarrow \boldsymbol{\emptyset}_{\boldsymbol{\epsilon}} \cup \mathbf{1}\, \mathsf{DS}^1_{\mathbf{length}}\, \boldsymbol{\emptyset}_{\boldsymbol{\epsilon}} \\ \sigma_{\mathbf{length}} &\rightarrow \sigma_{\mathbf{main}} \ \cup \ \boldsymbol{\emptyset}_{\boldsymbol{\epsilon}}\, \sigma_{\mathbf{length}} \end{aligned} \tag{3.1}$$

Information required for several applications can be posed as language recognition problems for this grammar. For example, During garbage collection, we may need to know whether a path in the heap, say **0010**, starting from the variable $\mathtt{u}$ in the root set is possibly live at program point π_6. This translates to the question of whether the language of D_π, after simplification using the function $\mathcal{S}$, contains **0010**. Formally, $\mathbf{0010} \in \mathcal{S}(\mathscr{L}(\mathsf{D}_\pi))$? Notice that we ask the membership question for strings belonging to $(\mathbf{0} + \mathbf{1})^*$, as these represent valid paths in the heap.

If the membership question was decidable, the dependence question would also be decidable. But, as we have already shown, the dependence question is undecidable and hence the membership question is also undecidable. The membership question can be shown to be undecidable in a similar way to the dependence question. Fortunately, the membership question becomes decidable if the grammars generated are regular. In the next section, we describe a method to safely over-approximate the CFGs generated from our analysis by regular grammars.

3.2.4 Over-approximating dependence grammars

We circumvent the problem of undecidability by over approximating the CFG by non-deterministic finite state automata (NFA) using the method of Mohri and Nederhof [63]. This method transforms a CFG G into a strongly regular grammar R such that $\mathscr{L}(G) \subseteq \mathscr{L}(R)$. This makes the membership question decidable at the cost of some precision.

Function createStronglyRegularGrammar(G)

Data: A context-free grammar G

Result: R, the strongly regular grammar over approximating G

create grammar R

/* M is the set of mutually recursive non-terminals in G */

$M \leftarrow \{A, B_1, B_2, ..., B_n\}$

add to R new non-terminals $\{\overline{A}, \overline{B_1}, \overline{B_2}, ..., \overline{B_n}\}$

foreach *(production* $P \in G$*)* **do**

add to R production $\overline{A} \rightarrow \epsilon$

/* α_i not empty */

if *(*P *is* $A \rightarrow \alpha_0 B_1 \alpha_1 B_2 \alpha_2 ... B_m \alpha_m$ *and* $m > 0$*)* **then**

add to R production $A \rightarrow \alpha_0 B_1$

add to R production $\overline{B_1} \rightarrow \alpha_1 B_2$

. . .

add to R production $\overline{B_m} \rightarrow \alpha_m \overline{A}$

else

add to R production $A \rightarrow \alpha_0 \overline{A}$

return R

Algorithm 4: Function to approximate a CFG by a strongly regular grammar [63].

If a CFG consists of a set of mutually recursive non-terminals such that the rules involved are not all left regular or not all right regular, then the method breaks the rules into right regular rules by introducing fresh non-terminals. For our example, the rule $\mathsf{D}^1_{\mathbf{length}}$ has a non-regular production $\mathbf{1}\mathsf{D}^1_{\mathbf{length}}\boldsymbol{\emptyset}_{\boldsymbol{\epsilon}}$. Algorithm 4 describes the procedure to over approximate a context free grammar by a strongly regular grammar. The steps for transforming these productions into right regular productions are:

1. Add a new non-terminal $\overline{\mathsf{D}^1_{\mathbf{length}}}$ to the grammar with the rule $\overline{\mathsf{D}^1_{\mathbf{length}}} \rightarrow \epsilon$.
2. Replace $\mathsf{D}^1_{\mathbf{length}} \rightarrow \boldsymbol{\emptyset}_{\boldsymbol{\epsilon}}$ by $\mathsf{D}^1_{\mathbf{length}} \rightarrow \boldsymbol{\emptyset}_{\boldsymbol{\epsilon}}\overline{\mathsf{D}^1_{\mathbf{length}}}$
3. Replace $\mathsf{D}^1_{\mathbf{length}} \rightarrow \mathbf{1}\mathsf{D}^1_{\mathbf{length}}\boldsymbol{\emptyset}_{\boldsymbol{\epsilon}}$ by $\mathsf{D}^1_{\mathbf{length}} \rightarrow \mathbf{1}\mathsf{D}^1_{\mathbf{length}}$ and $\overline{\mathsf{D}^1_{\mathbf{length}}} \rightarrow \boldsymbol{\emptyset}_{\boldsymbol{\epsilon}}\overline{\mathsf{D}^1_{\mathbf{length}}}$

The detailed algorithm and explanation of this approximation is described in Mohri

Input: A program P with a function **main** as entry point and $\sigma_{\mathbf{main}}$ as the user supplied demand

Output: A finite-state automaton ($\mathcal{FSM}$) for every program point π such that for every string $w \in (\mathbf{0}+\mathbf{1})^*$,
$\exists d\mathsf{prop}((\mathbf{main}), \sigma_{\mathbf{main}}, \pi, \delta)$ and $w \in \delta \Rightarrow w \in \mathscr{L}(\mathcal{FSM})$

Step 1 : (Section 3.2.1)

foreach (**define** $(f\ \overline{x})\ e_f(\overline{x})$) **do**

Obtain summary DS^i_f with respect to a symbolic demand σ resulting in equations :
$\mathsf{DS}^i_f(\sigma) = \ldots \mathsf{DS}^j_g(\sigma') \ldots$ /* $1 \leq i < \#Args(f)$ */

Step 2: (Section 3.2.2)

foreach e_f **do**

if f *is* **main then**

$\sigma_f = \sigma_{\mathbf{main}}$

else

$\sigma_f = \bigcup_{\pi' \in \Pi} \mathsf{DE}_{\pi'}$

where Π is the set of all program points where f is called in P

Compute demand $\mathcal{D}(e_f, \sigma_f, \mathbb{DS})$ to give DE_π at each $\pi\ :\ e$ in e_f

Step 3: (Section 3.2.3)

Obtain closed form, by expressing $\mathbb{DS}^i_f(\sigma)$ as $\mathsf{DS}^i_f\,\sigma$

DS^i_f is given by a CFG over $\{\mathbf{0}+\mathbf{1}+\bar{\mathbf{0}}+\bar{\mathbf{1}}+\emptyset_\epsilon\}$

Express all DE_π as CFG in terms of $\mathsf{DE}_{\pi'}$ and DS^i_f

Step 4: (Section 3.2.4)

foreach DE_π **do**

Convert DE_π to strongly regular grammar using Mohri-Nederhof transformation

Convert regular grammar to $\mathcal{FSM}$ and perform $\mathcal{S}$-simplification

Algorithm 5: Algorithm to convert the dependence analysis specification in Figure 3.1 to a computable form.

and Nederhof [63]. The rules for $\mathsf{D}^1_{\mathbf{length}}$ after the transformation are:

$$\begin{aligned} \mathsf{D}^1_{\mathbf{length}} &\rightarrow \emptyset_\epsilon \overline{\mathsf{D}^1_{\mathbf{length}}} \mid \mathbf{1}\mathsf{D}^1_{\mathbf{length}} \\ \overline{\mathsf{D}^1_{\mathbf{length}}} &\rightarrow \emptyset_\epsilon \overline{\mathsf{D}^1_{\mathbf{length}}} \mid \epsilon \end{aligned}$$

The strongly regular grammar is converted into a set of NFAs, one for each non-

terminal. The simplification is now done on the NFAs by repeatedly introducing ϵ edges to bypass pairs of consecutive edges labeled $\bar{\mathbf{0}}\mathbf{0}$ or $\bar{\mathbf{1}}\mathbf{1}$ and constructing the ϵ-closure until a fixed point is reached, after which the edges labeled $\bar{\mathbf{0}}$ and $\bar{\mathbf{1}}$ are deleted. The simplification does not change the semantics of the path in that the node reached by the path remains the same and no new edge is added to the path . The details of the algorithm to perform simplification on the NFAs, its correctness and termination proofs are given in [12, 44]. Finally, we remove all the edges labeled $\emptyset_{\epsilon}$ and convert the automaton into a deterministic automaton. These steps effectively implement the simplification function $\mathcal{S}$ rules for $\bar{\mathbf{0}}$ and $\bar{\mathbf{1}}$ to obtain forward access paths. This concludes our analysis procedure and these automata constitute the output of our dependence analysis. The entire procedure to bring the specification of dependence analysis to a computable form is shown in Algorithm 5.

3.3 Soundness of approximate dependence analysis

We now prove the correctness of our proposed dependence analysis. This involves showing that the demand computed by our analysis conservatively approximates the "real" demand on each expression that would meet the demand $\sigma_{\mathbf{main}}$ on the designated expression (**main**) in any execution of the program.

Consider the trace of a program in execution under DGS. Let δ represent the runtime or dynamic demand on an expression. Assume that an expression e appears on the trace for evaluation with an execution context $\mathcal{E} = (\rho, S, \_, \delta)$. The evaluation of e under the context $\mathcal{E}$ is deemed to be over, when its value v reaches the extent of evaluation specified by δ and is replaced by the continuation on the top of S. During this evaluation (of e under the context $\mathcal{E}$), consider a sub-expression e' of e that appears on the trace for evaluation with a context, say $\mathcal{E}' = (\rho', S', \_, \delta')$. Then the soundness of our analysis involves showing that if σ and σ' are the static demands on e and e' respectively, then $\delta \subseteq \sigma$ implies $\delta' \subseteq \sigma'$. In other words, if $\delta \subseteq \sigma$ and $\mathsf{prop}(\mathrm{e}, \delta, \mathrm{e'}, \delta')$, then $\delta' \subseteq \sigma'$. If this happens for every execution context $\mathcal{E}$ and every sub-expression e', we say that *the expression e preserves subsumption.* One can similarly talk about applications preserving subsumption.

We first show that every expression preserves the subsumption relation, provided applications, in particular function calls, preserve subsumption.

Lemma 3.2 *Assuming that applications preserve subsumption, expressions also preserves subsumption.*

PROOF. The proof is by induction on the structure of expressions. The base case is a **return** expression for which the proof is trivial. Now consider a **let** expression e given as (**let** $x \leftarrow s$ **in** e'). Assume as induction hypothesis that e' preserves the subsumption relation. Let e appear for evaluation on the trace of the program with a context carrying the demand δ. We assume that the static demand on e is σ and that $\delta \subseteq \sigma$. Then, by the rules of dependence analysis the static demand on e' is also σ. Further, the rules of DGS gives that e' appears on the trace with the context $(\rho \oplus \{x \mapsto \langle s, \rho \rangle\}, \_, \_, \delta)$, i.e. the dynamic demand on e' is also δ. So the premise of the induction hypothesis holds for e', and thus for each sub-expression of e' the subsumption relation holds. In particular, this subsumption relation will hold for the dynamic and static demands on x, if x appears on the trace for execution. i.e. if the dynamic and static demands on x are δ' and σ' respectively, then $\delta' \subseteq \sigma'$. Further, if x appears on the trace of the DGS execution with a demand δ', then in the next step of DGS, s also appears on the trace with the same dynamic demand δ'. By the dependence analysis rule for **let**, the static demand on s will include σ'. Since s is an application, because of the premise of the lemma, the subsumption relation will hold for the sub-expressions of s. Since the sub-expressions of e are made up of the sub-expressions of s and e', the result follows for **let**.

The case when e is an **if** expression directly follows from the induction hypothesis and the rules of dependence analysis and DGS. ■

We now have to show that the assumption regarding applications in Lemma 3.2 actually holds.

Lemma 3.3 *Applications preserve subsumption.*

PROOF. For an application that is not a function call, it is clear that the statement of the lemma holds. This is because such applications propagate the demand to their arguments in the same way in both the static analysis and the DGS.

To prove the lemma for a function call, say $(f\ x)$, we induct on the depth of the call. Assume that the function f is defined as (**define** $(f\ a)\ e_f$) and also assume that the evaluation context of $(f\ x)$ in the trace is $(\_, S, \_, \delta)$, the demand on the call in the static

analysis is σ, and $\delta \subseteq \sigma$. The DGS trace then evaluates e_f with a demand δ. Consider a static analysis of e_f with the demand σ. For the base case, assume that evaluation of e_f does not result in any more function calls. Therefore, by lemma 3.2, if the formal argument a comes on the trace with a demand δ' and the static demand on this occurrence of a is σ', we have $\delta' \subseteq \sigma'$. Since a is bound to (**id** x), the dynamic demand on x is δ' while the static demand is x is $\mathbb{DS}^1_f(\sigma)$. We have to show that $\delta' \subseteq \mathbb{DS}^1_f(\sigma)$. This directly follows from the rule DEMAND-SUMMARY.

For the inductive hypothesis, assume that all calls in e_f preserve subsumption. Due to the inductive hypothesis the premise of lemma 3.2 is satisfied and we can once again replay the argument of the base case and prove that if the dynamic demand on an occurrence of the argument a is δ' and the static demand is σ', we have $\delta' \subseteq \sigma'$. And for the reason as in the base case, we have $\delta' \subseteq \mathbb{DS}^1_f(\sigma)$, where $\mathbb{DS}^1_f(\sigma)$ is the static demand on x. ■

Theorem 3.4 *Consider the DGS trace of an arbitrary program. For any expression e that appears on the trace, the dynamic demand on e is subsumed by the static demand on e.*

PROOF. We first define the dynamic level of an expression e on the trace as follows. The only expression at level 0 is (**main**), and if a function call to f appears at level n, then each sub-expression e of the current incarnation of e_f that appears on the trace is at level $n+1$. The proof is by induction on the dynamic level n of expressions.

The base case $n = 0$ is trivial as both the dynamic and the static demands on (**main**) are the same, $\sigma_{\mathbf{main}}$. Now assume that the statement of the theorem holds for all expressions at level n. Consider any call $(f\ x)$ at level n. If the dynamic and static demands on $(f\ x)$ are δ and σ respectively, then from the induction hypothesis $\delta \subseteq \sigma$. While the dynamic demand on e_f is also δ, the static demand is σ_f which contains σ. Thus, the dynamic demand on e_f is subsumed by the static demand and therefore by Lemmas 3.2 and 3.3, for all sub-expressions of e_f that appear on the trace, the static demand subsumes the dynamic demand. Since this happens for all calls at dynamic level n, the theorem holds for all expressions at dynamic level $n+1$. ■

Chapter 4

Liveness-based garbage collection for lazy languages

Functional programs make extensive use of dynamically allocated memory. The allocation is either explicit (i.e., while using constructors) or implicit (while creating the runtime representations of unevaluated expressions, also called closures). Programs in lazy functional languages put additional demands on memory, as they require closures to be carried from the point of creation to the point of evaluation.

Although the runtime system of most functional languages includes a garbage collector to reclaim memory, empirical studies on Scheme [45] and Haskell [82] programs have shown that garbage collectors leave uncollected a large number of memory objects that are reachable but will assuredly not be used by the program later. This results in unnecessary retention of memory which can be safely garbage collected.

In this chapter, we propose the use of liveness analysis of heap cells to improve garbage collection (GC) in a lazy first-order functional language. Liveness analysis can identify cells which will definitely not be used by the program in future. By making this information available during garbage collection, these cells can be garbage collected, even if they are reachable. We use a modified version of the dependence analysis that was defined in chapter 2 to compute liveness information. The result of liveness analysis is an annotation of certain program points with deterministic finite-state automata (DFA), one for each variable in scope, capturing the liveness of the variables at these points. Depending on where GC is triggered, the collector consults a set of automata to restrict reachability during marking. This results in an increase in the garbage reclaimed and

consequently in fewer collections.

Whereas the idea of using static analysis to improve memory utilization has been shown to be effective for *eager* languages [12, 36, 41, 56], a straightforward extension of the technique is not possible for lazy languages, where heap-allocated objects may include closures. The additional complexity of replaying such techniques for lazy languages are as follows: Firstly, since data is made live by evaluation of closures, and in lazy languages the place in the program where this evaluation takes place cannot be statically determined, laziness complicates liveness analysis itself. Moreover, for liveness-based GC to be effective, we need to extend it to closures apart from evaluated data. Since a closure can escape the scope in which it was created, during garbage collection, it is not enough to refer to the liveness of only variables in scope. As we shall see later, we require closures to carry liveness information of its free variables. As a further optimization, as execution progresses and possible future uses are eliminated, we update the liveness information in a closure with a more precise version. For these reasons, the garbage collector also becomes significantly more complicated than a liveness-based collector for an eager language.

Experiments with a single generation copying collector (Section 4.4.3) confirm the expected performance benefits. Liveness-based collection results in an increase in garbage reclaimed. As a consequence, there is a reduction in the number of collections and a decrease in the minimum memory requirement. As an added benefit, there is also a reduction in the overall execution time in some of the benchmark programs.

4.1 Motivating example

Figure 4.1 shows an example in which the heap is represented by a graph in which a node either represents atomic values (**nil**, integers, etc.), or a **cons** cell containing **car** and **cdr** fields, or a closure (represented by shaded clouds). Edges in the graph are *references* and represent values of variables or fields. Figure 4.1(b) shows the lists `x` and `z` partially evaluated due to the **if** condition (**null**? (**car** `z`)). The edges shown by thick arrows are those which are live at π.

Thus, if a GC takes place at π with the heap shown in Figure 4.1(b), a liveness-based collector (LGC) will preserve only the cell referenced by `z` and the live cells constituting the closure referenced by (**cdr** `z`). In contrast, a reachability-based collector (RGC) will

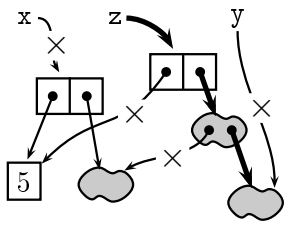
```
(define (length lst)
  (if (null? lst) 0 (+ 1 (length (cdr lst)))))

(define (append l1 l2)
  (if (null? l1) l2
   (cons (car l1) (append (cdr l1) l2))))

(define main)
  (let x ← (cons 5 (cons (cons 6 nil) nil) in
    (let y ← (cons 3 nil) in
      (let z ← (append x y) in
        (if (null? (car z)) 0 π: (length z))))))

(main)
```

(a) Example program.



(b) Memory graph at π.

denotes a closure. Thick edges denote live links. Traversal stops at edges marked × during garbage collection for a liveness-based collector.

Figure 4.1: Example Program and its Memory Graph

preserve all cells. In this chapter, we propose a static analysis of heap data that helps in determining the live references in the heap. Similar to dependence analysis, the result is a set of automata describing the liveness of variables at chosen program points. We also describe a GC scheme which uses the automata to collect the non-live areas of the heap during GC and implement a copying collector based on the scheme. Our experiments reveal interesting space-time trade-offs in the engineering of the collector—for example, updating liveness information carried in closures during execution results in more garbage being collected.

Premise	Transition	Rule name
	$\rho,\ (\rho', \ell, e){:}S,\ \mathbf{H},\ \kappa \rightsquigarrow \rho',\ S,\ \mathbf{H}[\ell := \kappa],\ e$	CONST
$\rho(x)$ is $\langle s, \rho' \rangle$	$\rho, S, \mathbf{H}, x \rightsquigarrow \rho', S, \mathbf{H}, s$	VAR
	$\rho,\ (\rho', \ell, e){:}S,\ \mathbf{H},\ (\mathbf{cons}\ x\ y) \rightsquigarrow$ $\rho',\ S,\ \mathbf{H}[\ell := (\rho(x), \rho(y))],\ e$	CONS
$\mathbf{H}(\rho(x))$ is (v, d)	$\rho,\ (\rho', \ell, e){:}S,\ \mathbf{H},\ (\mathbf{car}\ x) \rightsquigarrow$ $\rho',\ S,\ \mathbf{H}[\ell := v],\ e$	CAR-SELECT
$\mathbf{H}(\rho(x))$ is $(\langle s, \rho' \rangle, d)$	$\rho,\ S,\ \mathbf{H},\ (\mathbf{car}\ x) \rightsquigarrow$ $\rho',\ (\rho, addr(\langle s, \rho' \rangle), (\mathbf{car}\ x)){:}S,\ \mathbf{H},\ s$	CAR-1-CLO
$\mathbf{H}(\rho(x))$ is $\langle s, \rho' \rangle$	$\rho,\ S,\ \mathbf{H},\ (\mathbf{car}\ x) \rightsquigarrow$ $\rho',\ (\rho, \rho(x), (\mathbf{car}\ x)){:}S,\ \mathbf{H},\ s$	CAR-CLO
$\mathbf{H}(\rho(x)), \mathbf{H}(\rho(y)) \in \mathbb{N}$	$\rho,\ (\rho', \ell, e){:}S,\ \mathbf{H},\ (+\ x\ y) \rightsquigarrow$ $\rho',\ S,\ \mathbf{H}[\ell := \mathbf{H}(\rho'(x)) + \mathbf{H}(\rho'(y))],\ e$	PRIM-ADD
$\mathbf{H}(\rho(x)) \notin \mathbb{N}$	$\rho,\ S,\ \mathbf{H},\ (+\ x\ y) \rightsquigarrow$ $\rho',\ (\rho, \rho(x), (+\ x\ y)){:}S,\ \mathbf{H},\ x$	PRIM-1-CLO
$\mathbf{H}(\rho(y)) \notin \mathbb{N}$	$\rho,\ S,\ \mathbf{H},\ (+\ x\ y) \rightsquigarrow$ $\rho', (\rho,\ \rho(y),\ (+\ x\ y)){:}S,\ \mathbf{H},\ y$	PRIM-2-CLO
f defined as $(\mathbf{define}\ (f\ \vec{y})\ e_f)$	$\rho,\ S,\ \mathbf{H},\ (f\ \vec{x}) \rightsquigarrow [\vec{y} \mapsto \rho(\vec{x})],\ S,\ \mathbf{H},\ e_f$	FUNCALL
ℓ is a new location	$\rho,\ S,\ \mathbf{H},\ (\mathbf{let}\ x \leftarrow s\ \mathbf{in}\ e) \rightsquigarrow$ $\rho \oplus [x \mapsto \ell],\ S,\ \mathbf{H}[\ell := \langle s, \lfloor \rho \rfloor_{FV(s)} \rangle],\ e$	LET
$\mathbf{H}(\rho(x)) \neq 0$	$\rho,\ S,\ \mathbf{H},\ (\mathbf{if}\ x\ e_1\ e_2) \rightsquigarrow \rho,\ S,\ \mathbf{H},\ e_1$	IF-TRUE
$\mathbf{H}(\rho(x)) = 0$	$\rho,\ S,\ \mathbf{H},\ (\mathbf{if}\ x\ e_1\ e_2) \rightsquigarrow \rho,\ S,\ \mathbf{H},\ e_2$	IF-FALSE
$\mathbf{H}(\rho(x)) = \langle s, \rho' \rangle$	$\rho,\ S,\ \mathbf{H},\ (\mathbf{if}\ x\ e_1\ e_2) \rightsquigarrow$ $\rho',\ (\rho, \rho(x), (\mathbf{if}\ x\ e_1\ e_2)){:}S,\ \mathbf{H},\ x$	IF-CLO
	$\rho,\ S,\ \mathbf{H},\ (\mathbf{return}\ x) \rightsquigarrow$ $\rho',\ (\rho, \rho(x), (\mathbf{return}\ x)){:}S,\ \mathbf{H},\ x$	RETURN

Figure 4.2: A small-step semantics for the language.

4.1.1 Semantics

We now give a small-step semantics for the language described in Section 2.3. We first specify the domains used by the semantics:

$$\begin{array}{rcll} \mathbf{H} : \mathit{Heap} & = & \mathit{Loc} \rightarrow (\mathit{Data} + \{\mathit{empty}\}) & \text{– Heap} \\ d : \mathit{Data} & = & \mathit{Val} + \mathit{Clo} & \text{– Values \& Closures} \\ v : \mathit{Val} & = & \mathbb{N} + \{\mathbf{nil}\} + \mathit{Data} \times \mathit{Data} & \text{– Values} \\ c : \mathit{Clo} & = & (\mathit{App} \times \mathit{Env}) & \text{– Closures} \\ \rho : \mathit{Env} & = & \mathit{Var} \rightarrow \mathit{Loc} & \text{– Environment} \end{array}$$

Here *Loc* is a countable set of locations in the heap. A non-empty location either contains a *closure*, or a value in Weak Head Normal Form (WHNF)[74]. For our implementation, a value in WHNF is either a number, or the empty list **nil** or a **cons** cell with possibly unevaluated constituents. A closure is a pair $\langle s, \rho \rangle$ in which s is an unevaluated application, and ρ maps free variables of s to their respective locations. Since all data objects are boxed, we model an environment as a mapping from the set of variables of the program *Var* to locations in the heap.

The semantics of expressions (and applications[1]) are given by transitions of the form $\rho, S, \mathbf{H}, e \rightsquigarrow \rho', S', \mathbf{H}', e'$. Here S is a stack of continuation frames. Each continuation frame is a triple (ρ'', ℓ, e_{next}), signifying that the location ℓ has to be updated with the value of the currently evaluating expression and e_{next} is to be evaluated next in the environment ρ''. The initial state of the transition system is:

$$([\,]_\rho,\ (\rho_{init},\ \ell_{\mathbf{ans}},\ (\mathbf{evalAndPrint\ ans})) : [\,]_S, [\,]_{\mathbf{H}},\ (\mathbf{main}))$$

in which $[\,]_\rho$, $[\,]_H$ and $[\,]_S$ are the empty environment, heap and stack respectively. The initial stack consists of a single continuation frame in which **ans** is a distinguished variable that will eventually be updated with the value of (**main**), and ρ_{init} maps **ans** to a location $\ell_{\mathbf{ans}}$. As is customary for lazy languages, the result of evaluation of (**main**) is in WHNF. Full evaluation is achieved through interaction with a printing mechanism modelled as a function **evalAndPrint** which evaluates the unevaluated parts of (**main**), in case (**main**) is a structure. This is a standard runtime support assumption for lazy languages [74]. The operator : pushes elements on top of the stack.

[1] In most contexts, we shall use the term 'expression' and the notation e to stand for both expressions and applications.

The notation $[\vec{x} \mapsto \vec{\ell}]$ represents an environment that maps variables x_i to locations ℓ_i and $\mathbf{H}[\ell := d]$ indicates an update of $\mathbf{H}$ at ℓ with d. $\rho \oplus \rho'$ represents the environment ρ shadowed by ρ' and $\lfloor \rho \rfloor_X$ represents the environment restricted to the variables in X. Finally $FV(s)$ represents the free variables in the application s and $addr(c)$ gives the address of the closure c in the heap. As a convention, we use d to represent a data value which may either be in WHNF or a closure and v to represent values which are always in WHNF.

The small-step semantics is shown in Figure 4.2. Unlike an eager language, evaluation of a let expression (**let** $x \leftarrow s$ **in** e) does not result in the evaluation of s. Instead, as the LET rule shows, a closure is created and bound to x. The program points which trigger the evaluation of these closures are an **if** condition (IF-CLO) and a **return** (RETURN-CLO). We call such points *evaluation points* (ep) and label them with ψ instead of π. As an example of closure evaluation, we explain the three rules for (**car** x). If x is a closure, it is evaluated to WHNF, say (d_1, d_2). This is given by the rule CAR-CLO. If d_1 is not in WHNF, it is also evaluated (CAR-1-CLO). The address to be updated with the evaluated value is recorded in a continuation frame. This is required for the evaluation to be lazy, else d_1 may be evaluated more than once due to sharing [74]. Only after this is the actual selection done (CAR-SELECT).

4.2 Liveness

A variable is *live* if there is a possibility of its value being used in future computations and dead if it is definitely not used. Classical liveness analysis models liveness as a boolean value—a variable is either live or it is dead. Heap-allocated data needs a richer model than classical liveness—a model which talks about liveness of references possibly pointing to structured data. As an example, consider a list x. Assume that, at a program point, future computations only refer up to the third member of x. A precise liveness model should be able to clearly capture such liveness values. Liveness, in this sense, signifies future accesses of parts of a structure, and therefore the notion of access paths that was introduced in Chapter 2 can be used as its natural representation. For example, the liveness of the list x mentioned earlier can be represented as the set of access paths $\{\mathbf{0}, \mathbf{10}, \mathbf{110}\}$. However, the notion of access paths (or what it represents) has to be modified to account for structures

which are not fully evaluated. We shall provide the semantics of access path in terms of heap accesses later, especially in the context of a lazy language.

In contrast to dependence, liveness is a property that is applicable only to variables. Since a liveness-based garbage collection is guided by the liveness of variables, it has to be computed (or, at the least, approximated) for each program point where a garbage collector could be potentially invoked. While our method is not restricted to a particular garbage collection mechanism, we explain our method using a copying collector [20, 26]. Whenever a garbage collection is triggered, starting with the root set (variable references on the stack), the garbage collector consults the liveness value associated with the variable and copies only the parts of the value which are live.

We now connect liveness with dependence analysis. For computing liveness, we are interested in the entire output, hence the designated expression is (**main**) and the effective demand on (**main**) is $(\mathbf{0}+\mathbf{1})^*$ which we also refer as σ_{all}. We reiterate the point made in Section 2.4.1 that the demand of σ_{all} on (**main**) is achieved indirectly through the repeated invocation of **evalAndPrint**. Dependence analysis propagates this demand to every expression, however in the context of liveness analysis we would be interested in the demands on variables only. The demand on a variable occurrence gives the use of the variable's value in computing the result of the focus expression. On the other hand, the liveness of a variable at a program point π takes into account all the future uses of the variable beyond π. Therefore, liveness of a variable at a program point is the union of demands of all occurrences of the variable beyond the program point. As an example, consider the liveness of the variable `xs` in program 4.3 between the program points π_1 and π_3. There are two occurrences of `xs` beyond π_1, in (**append** `xs ys`) at π_3 and (**car** `xs`) at π_6. The liveness of `xs` between π_1 and π_3 is the union of demands on both these occurrences. Now, consider the liveness of `xs` at π_4. The only use of `xs` beyond π_4 is (**car** `xs`) at π_6. Hence, the liveness at π_4 is given by the demand on `xs` at π_6. This describes the liveness of *stack variables*, i.e. the function arguments and the variables defined in a **let** (also called the root set). We call liveness of such variables *stack-liveness* to distinguish it from the liveness of closure variables which we will introduce shortly.

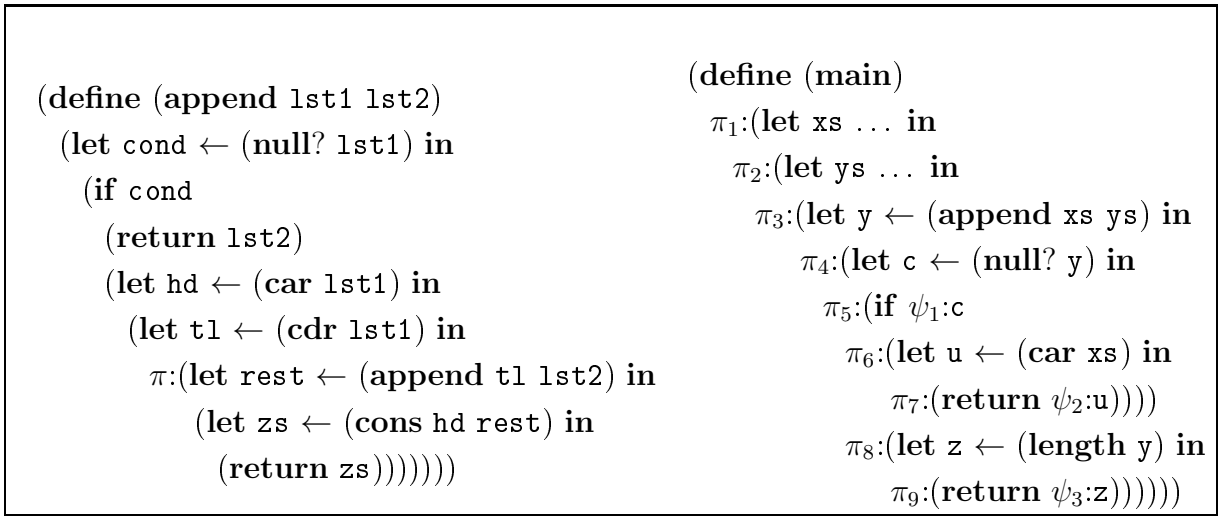


Figure 4.3: Example illustrating liveness of closures

4.2.1 Liveness analysis for lazy languages

In an eager language, the order of evaluation of the expression (**let** $x \leftarrow s$ **in** π:e) is as follows: s is evaluated first and its value is bound to x and then e is evaluated. Now consider a variable y that occurs free in s, and consider the program point π just before e. Since s has already been evaluated, the occurrence of y in s does not contribute to the liveness of the stack variable y at π. In general, the demand on any variable which is part of s need not be considered in computing liveness of the corresponding stack variable at or beyond this program point. As an example, in Figure 4.3, it can be seen from the program text itself that (**append** `xs ys`) would be evaluated before π_4, and hence liveness of the stack variable `xs` need not consider the demand generated by the use of `xs` in the expression (**append** `xs ys`). Further, as described in [12], during garbage collection, stack variables always point to fully evaluated values and hence it is sufficient to consider stack-liveness while doing liveness-based garbage collection of eager languages.

In contrast, a lazy evaluation of the expression (**let** `x` $\leftarrow$ s **in** e) creates a closure for `s` instead of evaluating it and binds this closure to x. In a lazy language statically determining the order of evaluation of closures is not possible and hence, it is not possible to determine statically whether a variable is bound to a closure or to an evaluated value at a given program point. Thus, during liveness-based garbage collection for lazy languages, a stack variable may point to an evaluated value or a closure. If the value is fully evaluated, we can just use the stack-liveness to garbage collect it, but if it is a closure then stack-liveness cannot be used directly to garbage collect the closure. This is illustrated in

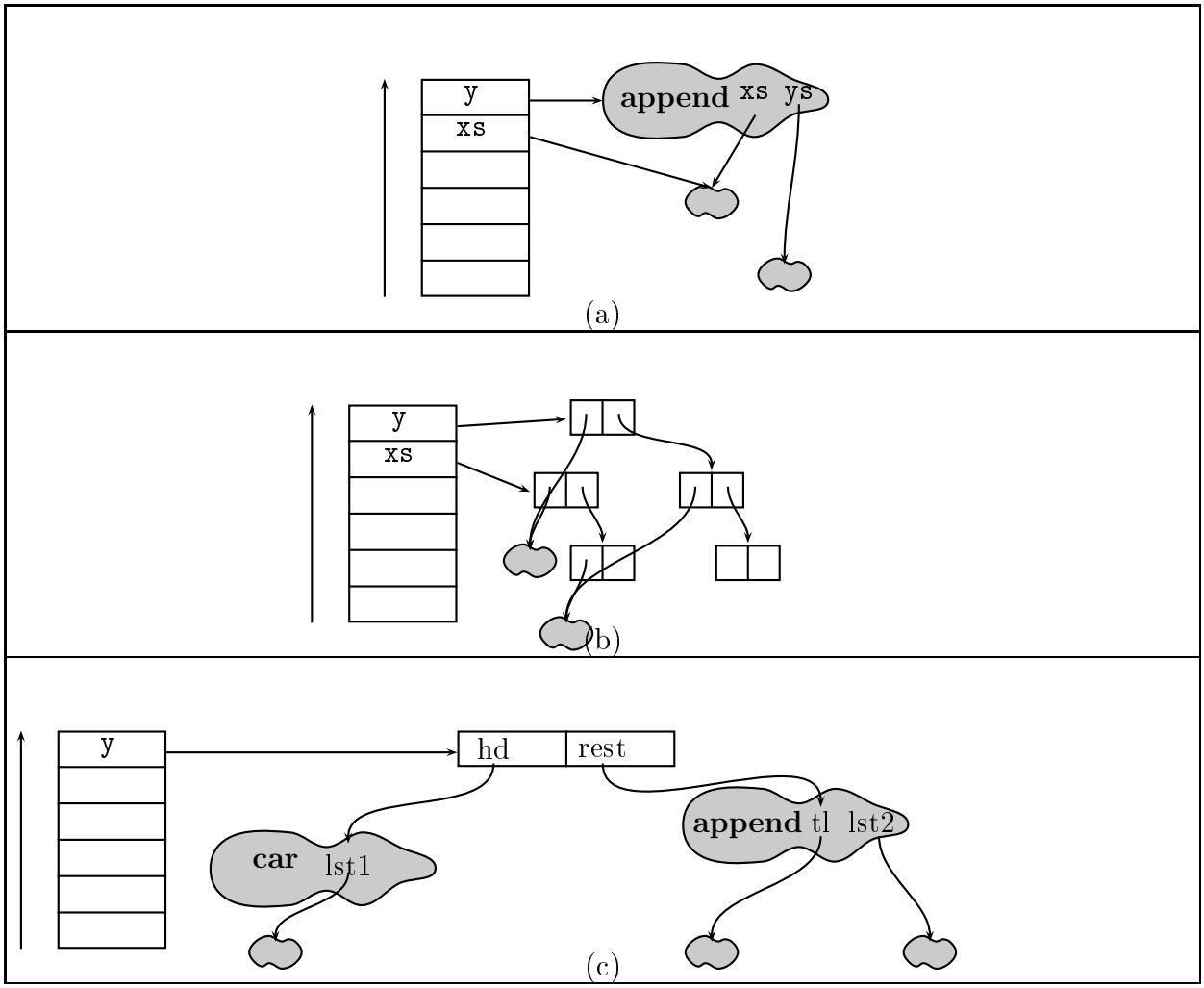


Figure 4.4: Different liveness situations encountered during garbage collection of `y` in a lazy language, (a)Liveness at π_4 when `y` is a closure (b) Liveness at ψ_3 when the spine of `y` is evaluated (c) Liveness at π_8 where `y` points to a **cons** cell containing references to `rest` and `hd` declared in function **append**

Figure 4.3, where determining whether (**append** `xs ys`) would be evaluated before π_4 is not possible. Therefore, if the garbage collector encounters the closure corresponding to (**append** `xs ys`) at π_4, there are two alternatives: either to treat the closure as useful data and copy it, or to do a liveness-based garbage collection on the closure itself. The latter can result in some more space being reclaimed.

Figure 4.4 depicts the scenarios described earlier. Figure 4.4(a) shows the situation where `y` has not been evaluated, i.e. it is a closure. Notice that the memory corresponding to `xs` has references from the stack and also from the closure containing it. Figure 4.4(b) depicts the situation when `y` is fully evaluated. In this case, the future use of y is fully

accounted for by its stack-liveness, and this can be used for garbage collecting `y` at π_4.

There is yet another reason why stack-liveness alone is not sufficient for liveness-based garbage collection of lazy languages. In a lazy language, data constructors (for example, **cons**) are lazy, i.e. they do not evaluate their arguments. Therefore, when a **cons** cell is returned from a function, it might contain closures. These closures in turn hold references to variables which may be defined locally in the function returning the **cons**. Let us consider a garbage collection after the **cons** is returned. When the **cons** cell is being garbage collected, the current root set has only variables which are in the current scope. Since the references inside the **cons** cell were defined in a scope which is no longer on the program stack it is not possible to determine the liveness of the references inside the **cons** cell. In Figure 4.3, the variable `rest` is defined locally in the function **append**. This variable escapes from the scope when returned as part of the **cons** cell `zs`. In any garbage collection triggered beyond this return point, none of the root sets contains a reference to the variable `rest` which is part of the returned **cons** cell. Thus, determining the liveness of the reference becomes impossible. This situation is shown in Figure 4.4(c).

The solution to both the challenges is to treat variables which are part of a closure as first class citizens from the point of view of garbage collection, and treat them as being separate from the variables introduced by **let**s or function arguments. We call such variables as *closure variables* and consider liveness of both stack variables and closure variables during garbage collection. It is important to clarify that a stack variable and its corresponding closure variables are different references to the same memory location.

Notationally, a closure variable is distinguished from its corresponding stack variable by subscripting it with the label of the program point where the closure was created. As an example, in Figure 4.3, the closure variables $\mathtt{y}_{\pi_4}$ and $\mathtt{y}_{\pi_8}$ correspond to the stack variable `y` defined at π_3 [2]. Each closure carries liveness information of the variables which are part of a closure within the closure itself. Liveness of a closure variable is exactly the demand on that particular occurrence of the variable and hence is different from stack-liveness. For example, the closure-liveness of `xs` is just the demand on `xs` due to its use in the evaluation of the expression (**append** `xs ys`) in the context of the demand on the expression. The closure for (**append** `xs ys`) needs to carry this liveness information of

[2]Multiple occurrences of the same variable in an application are further distinguished by their positions in the application.

`xs` and `ys` within itself. In the modified garbage collection scheme, if the reference being garbage collected originates from the stack, the corresponding stack-liveness is used to garbage collect it and if the reference originates inside a closure the closure-liveness is used. While garbage collecting a reference on the stack, if we encounter a closure, the closure arguments are treated as references from which garbage collection, guided by the closure-liveness of the references, is initiated.

In summary, the major differences between the formulations of liveness-based garbage collection of lazy languages and eager languages [12] are:

1. Introducing the notion of closure variable and treating them as first-class citizens from the perspective of garbage collection.
2. The association of liveness with closure variables.
3. Handling evaluated values and closures differently during garbage collection.

We have been using access paths to represent liveness of variables. Assume that a garbage collection is triggered at π, where a variable x has a liveness α. In a lazy language, x may point to a closure or to a structure that may contain closures. While α still represents the set of accesses that might be performed in future when the closure is fully evaluated, a garbage collection triggered at π would use α to access only the evaluated parts of the structure, till a closure is encountered. Beyond those points, the garbage collector uses the liveness values of the closure variables to do the garbage collection. Formally, given an initial location ℓ (usually a reference corresponding to a variable) and a heap $\mathbf{H}$, semantically an access path α represents a reference, denoted $\mathbf{H}[\![\ell, \alpha]\!]$, in the heap that is obtained by starting with ℓ and chasing the **car** or **cdr** fields in the heap as specified by the access path. $\mathbf{H}[\![\ell, \alpha]\!]$ denotes the liveness of the heap rooted at ℓ only if the path followed in the heap is *closure-free*. If this path is intercepted by a closure, say (**car** y_π), then the liveness of the path starting from y_π is given by the demand on y_π. As we shall see in Section 4.4.1, the liveness of the closure variable y_π is recorded along with the closure for s so that the GC can refer to it during garbage collection.

4.2.2 The analysis

Figure 4.5 shows the dependence analysis introduced in Section 3.1 modified for liveness. Unlike demands which can be associated with both variables or expressions, liveness is always associated with variables. Hence, we modify the rules of dependence analysis to

$$\boxed{\mathcal{A} :: (\mathsf{App}, \mathsf{Demand}, \mathsf{FuncSummaries}) \to \mathsf{LivenessEnvironment}}$$

$$\begin{aligned}
\mathcal{A}(\pi{:}\kappa, \sigma, \mathbb{DS}) &= \{\}, \text{ for constants including } \mathbf{nil} \\
\mathcal{A}(\pi{:}(\mathbf{null?}\ x), \sigma, \mathbb{DS}) &= \{x \mapsto \boldsymbol{\emptyset}_{\boldsymbol{\epsilon}}\sigma\} \\
\mathcal{A}(\pi{:}(+\ x\ y), \sigma, \mathbb{DS}) &= \{x \mapsto \boldsymbol{\emptyset}_{\boldsymbol{\epsilon}}\sigma\} \cup \{y \mapsto \boldsymbol{\emptyset}_{\boldsymbol{\epsilon}}\sigma\} \\
\mathcal{A}(\pi{:}\ (\mathbf{car}\ x), \sigma, \mathbb{DS}) &= \{x \mapsto \mathbf{0}\sigma\} \\
\mathcal{A}(\pi{:}\ (\mathbf{cdr}\ x), \sigma, \mathbb{DS}) &= \{x \mapsto \mathbf{1}\sigma\} \\
\mathcal{A}(\pi{:}(\mathbf{cons}\ x\ y), \sigma, \mathbb{DS}) &= \{x \mapsto \bar{\mathbf{0}}\sigma\} \cup \{y \mapsto \bar{\mathbf{1}}\sigma\} \\
\mathcal{A}(\pi{:}(f\ y_1\ \cdots\ y_n), \sigma, \mathbb{DS}) &= \textstyle\bigcup_{i=1}^{n}\{y_i \mapsto \mathbb{DS}_f^i(\sigma)\}
\end{aligned}$$

$$\boxed{\mathcal{D} :: (\mathsf{Exp}, \mathsf{Demand}, \mathsf{FuncSummaries}) \to \mathsf{LivenessEnvironment}}$$

$$\begin{aligned}
\mathcal{D}(\pi{:}(\mathbf{return}\ x), \sigma, \mathbb{DS}) &= \mathcal{L}_\pi \\
&\quad \text{where } \mathcal{L}_\pi = \{x \mapsto \sigma\} \\
\mathcal{D}(\pi{:}(\mathbf{if}\ x\ \pi_1{:}e_1\ \pi_2{:}e_2), \sigma, \mathbb{DS}) &= \mathcal{L}_{\pi_1} \cup \mathcal{L}_{\pi_2} \cup \{x \mapsto \boldsymbol{\emptyset}_{\boldsymbol{\epsilon}}\sigma\} \\
&\quad \mathcal{L}_{\pi_1} = \mathcal{D}(\pi_1{:}e_1, \sigma, \mathbb{DS}) \\
&\quad \mathcal{L}_{\pi_2} = \mathcal{D}(\pi_1{:}e_2, \sigma, \mathbb{DS}) \\
\mathcal{D}(\pi{:}(\mathbf{let}\ x\ \leftarrow\ \pi_1{:}s\ \mathbf{in}\ \pi_2{:}e), \sigma, \mathbb{DS}) &= \mathcal{L}_{\pi_2} \setminus x.* \cup \mathcal{L}_{\pi_1} \\
&\quad \mathcal{L}_{\pi_1} = A(\pi_1{:}s, \sigma', \mathbb{DS}) \\
&\quad \sigma' = \mathcal{L}_{\pi_2}(x) \\
&\quad \mathcal{L}_{\pi_2} = \mathcal{D}(\pi_2{:}e, \sigma, \mathbb{DS}) \\
&\quad \text{where } x.* \text{ represents all access paths starting from } x
\end{aligned}$$

$$\boxed{\mathbb{DS} \in \mathsf{FuncSummaries} :: \mathsf{Funcname} \to (\mathsf{Demand} \to (\mathsf{Demand}_1, \ldots, \mathsf{Demand}_n))}$$

$$\frac{\forall f, \forall i, \forall \sigma :\ \ \mathcal{D}(\pi : e_f, \sigma, \mathbb{DS}) = \mathcal{L}_\pi, \mathbb{DS}_f^i = \bigcup_{\pi \in \Pi}\ \mathcal{L}_\pi(z_i)}{df_1 \ldots df_k\ \vdash^l\ \mathbb{DS}}\ (\textsc{function-summaries})$$

$(\mathbf{define}\ (f\ z_1\ \ldots\ z_n)\ \ e_f)$ is one of $df_1 \ldots df_k$, $1 \le i \le n$,

and Π represents all occurrences of z_i in e_f

Figure 4.5: Dependence analysis modified to compute liveness

now compute demands only for variables. While a demand environment was a mapping from a program-point to a demand, a *liveness environment* is a mapping from a variable to a demand. It is often expressed as a set, for example by writing $\{\mathtt{x}.\mathbf{11}, \mathtt{y}.\mathbf{1}, \mathtt{z}.\mathbf{0}\}$ instead of $[\mathtt{x} \mapsto \{\mathbf{11}\}, \mathtt{y} \mapsto \{\epsilon\}, \mathtt{z} \mapsto \{\mathbf{0}\}]$. The function $\mathcal{A}$ takes an application s and a demand σ and returns a liveness environment that maps the free variables of s to sets of access paths representing their closure-liveness. The closure-liveness is stored as part of

the closure itself, and consulted while exploring the heap starting from closure variables during garbage collection. The function $\mathcal{D}$ uses $\mathcal{A}$ to propagate liveness across expressions and computes program-point-wise stack-liveness. We use σ to range over demands, α to range over access paths and $\mathcal{L}_\pi$ to denote the liveness environment at program point π. The liveness of an individual variable y at program point π is $\mathcal{L}_\pi(y)$.

In a lazy language, an expression is not evaluated unless required. Our analysis captures this by ensuring that no liveness generated is independent of the incoming demand. The $\emptyset_\epsilon$ symbol ensures that an ϵ liveness is generated only if the incoming demand is non-null. Function calls are handled as in dependence analysis, using the third parameter $\mathbb{DS}$ that represents the summaries of all functions in the program. In case of liveness also, we prefer the least solution as it ensures the safe collection of the greatest amount of garbage.

The major modification in the dependence analysis rule happens in the rule for **let**. To understand the liveness rule for π:(**let** $x \leftarrow \ \pi_1$:s **in** π_2:e), observe that the value of **let** is the value of its body e. Thus the liveness environment $\mathcal{L}_{\pi_2}$ of e is calculated for the given demand σ. The stack variable x gets its liveness from the liveness environment of e and this liveness is transferred to s generating closure-liveness of the variables of s. Finally, the liveness environment at π is computed by killing the stack-liveness of x, and taking a union of the closure-liveness in s and the stack-liveness in e. Apart from the fact that liveness is computed only for variables, killing of stack-liveness is the primary difference between dependence analysis and liveness analysis.

4.3 An example

We now use an example program to show liveness computation and the differences between stack-liveness and closure-liveness. Let us consider the liveness of variable `xs` due to the evaluation point ψ_3.

1. The stack-liveness of `xs` just before executing the expression at π_3 is due to its uses in the expressions (**append** `xs ys`), (**car** `xs`) and is $\mathbb{DS}^1_{\mathbf{append}}(\mathbb{DS}^1_{\mathbf{length}}(\sigma_{all})) \cup \mathbf{0}\sigma_{all}$. However at π_5, the liveness is $\mathbf{0}\sigma_{all}$ as the liveness no longer includes the use of `xs` in (**append** `xs ys`).
2. In contrast, closure-liveness of `xs` at π_3 is only due to its use in (**append** `xs ys`).

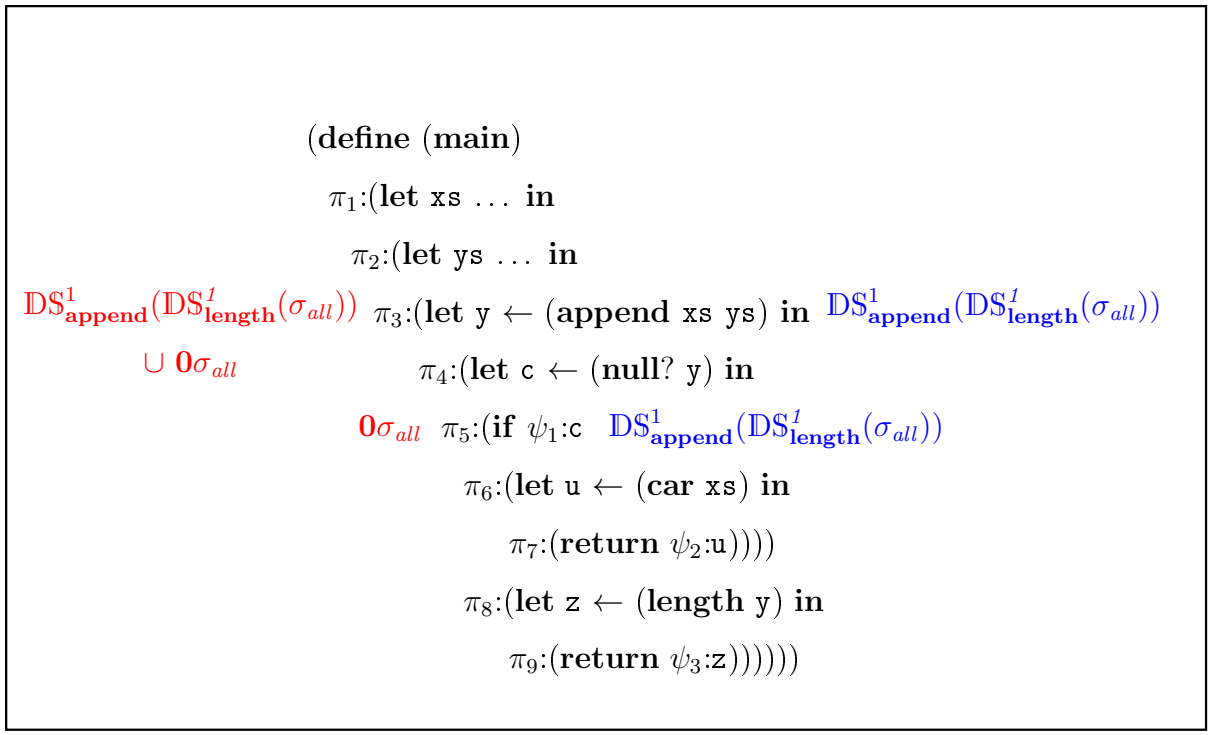


Figure 4.6: Stack and closure liveness for variable `xs` at program points π_3 and π_5. Stack liveness is indicated in red and Closure liveness in blue. Stack liveness changes between π_3 and π_5 as the expression (**append** `xs ys`) is not considered at π_5 for liveness computation. Closure variable remains unchanged.

Closure-liveness of `xs` due to the evaluation point ψ_3 is computed by starting with the expression at ψ_3 and transferring demands via the expressions (**length** `y`), till we reach `xs` at π_3. Notice, the expression (**car** `xs`) is not considered as the expression is not evaluated along the path starting from ψ_3.

4.4 Computing liveness information

We now use the liveness of the closure variable $\mathtt{xs}_{\pi_3}$ in example 4.1 to illustrate liveness computation. The liveness of the variable $\mathtt{xs}_{\pi_3}$ is determined by how the function **append** uses its first argument and how the stack variable `y` is used in the program. The liveness of the stack variable `y` is given by the union of the liveness of the closure variables $\mathtt{y}_{\pi_4}$ and $\mathtt{y}_{\pi_8}$. To compute the liveness value, we require the demand transformers for the user-defined functions **append** and **length**. The demand transformers of **append** and **length** are given by:

$$\mathbb{DS}^1_{\mathbf{length}}(\sigma) = \boldsymbol{\emptyset_\epsilon}\sigma \cup \mathbf{1}\mathbb{DS}^1_{\mathbf{length}}(\boldsymbol{\emptyset_\epsilon}\sigma)$$
$$\mathbb{DS}^1_{\mathbf{append}}(\sigma) = \boldsymbol{\emptyset_\epsilon}\sigma \cup \mathbf{0\bar{0}}\sigma \cup \mathbf{1}\mathbb{DS}^1_{\mathbf{append}}(\bar{\mathbf{1}}\sigma)$$

We now use the property that all rules of our analysis always prefix σ with symbols to rewrite $\mathbb{DS}^i_f(\sigma)$ as $\mathsf{DS}^i_f\,\sigma$ (DS^i_f concatenated with σ). After the rewrite we can cancel out σ from both the LHS and RHS to get the modified equations for $\mathbb{DS}_{\mathbf{length}}$ and $\mathbb{DS}_{\mathbf{append}}$:

$$\mathsf{D}^1_{\mathbf{length}} = \boldsymbol{\emptyset_\epsilon} \cup \mathbf{1}\mathsf{D}^1_{\mathbf{length}}\boldsymbol{\emptyset_\epsilon}$$
$$\mathsf{D}^1_{\mathbf{append}} = \boldsymbol{\emptyset_\epsilon} \cup \mathbf{0\bar{0}} \cup \mathbf{1}\mathsf{D}^1_{\mathbf{append}}\bar{\mathbf{1}}$$

Viewing these equation as a CFGs with $\{\mathbf{1}, \mathbf{1}, \bar{\mathbf{0}}, \bar{\mathbf{1}}, \boldsymbol{\emptyset_\epsilon}\}$ as terminal symbols and $\mathsf{D}^1_{\mathbf{length}}$ and $\mathsf{D}^1_{\mathbf{append}}$ as non-terminals, we get the following productions:

$$\begin{aligned}
\mathsf{D}^1_{\mathbf{length}} &\rightarrow \boldsymbol{\emptyset_\epsilon} \mid \mathbf{1}\mathsf{D}^1_{\mathbf{length}}\boldsymbol{\emptyset_\epsilon} \\
\mathsf{D}^1_{\mathbf{append}} &\rightarrow \boldsymbol{\emptyset_\epsilon} \mid \mathbf{0\bar{0}} \mid \mathbf{1}\mathsf{D}^1_{\mathbf{append}}\bar{\mathbf{1}}
\end{aligned}$$

The liveness of $\mathtt{xs}_{\pi_3}$ is given by the equation:

$$\mathcal{L}_{\pi_3}(\mathtt{xs}) \rightarrow \boldsymbol{\emptyset_\epsilon}\sigma_{all} \mid \mathsf{D}^1_{\mathbf{append}}\mathsf{D}^1_{\mathbf{length}}\boldsymbol{\emptyset_\epsilon}\sigma_{all}$$

Both $\mathsf{D}^1_{\mathbf{length}}$ and $\mathsf{D}^1_{\mathbf{append}}$ contain context-free productions. Hence, we perform the Mohri-Nederhof transformation on these non-terminals to convert them to strongly regular grammars. The grammar post conversion is:

$$\begin{aligned}
{\mathsf{D}^1_{\mathbf{length}}}' &\rightarrow \boldsymbol{\emptyset_\epsilon}{\mathsf{D}^1_{\mathbf{length}}}' \mid \epsilon \\
\mathsf{D}^1_{\mathbf{length}} &\rightarrow \mathbf{1}\mathsf{D}^1_{\mathbf{length}} \mid \boldsymbol{\emptyset_\epsilon}{\mathsf{D}^1_{\mathbf{length}}}' \\
{\mathsf{D}^1_{\mathbf{append}}}' &\rightarrow \bar{\mathbf{1}}{\mathsf{D}^1_{\mathbf{append}}}' \mid \epsilon \\
\mathsf{D}^1_{\mathbf{append}} &\rightarrow \mathbf{0\bar{0}}{\mathsf{D}^1_{\mathbf{append}}}' \mid \boldsymbol{\emptyset_\epsilon}{\mathsf{D}^1_{\mathbf{append}}}' \mid \mathbf{1}\mathsf{D}^1_{\mathbf{append}}
\end{aligned}$$

We now construct the automaton corresponding to $\mathtt{xs}_{\pi_3}$ using the strongly regular grammars. The automaton and the corresponding simplifications are shown in Figure 4.7. When no more simplification rules are applicable, all the bar-edges are dropped from the

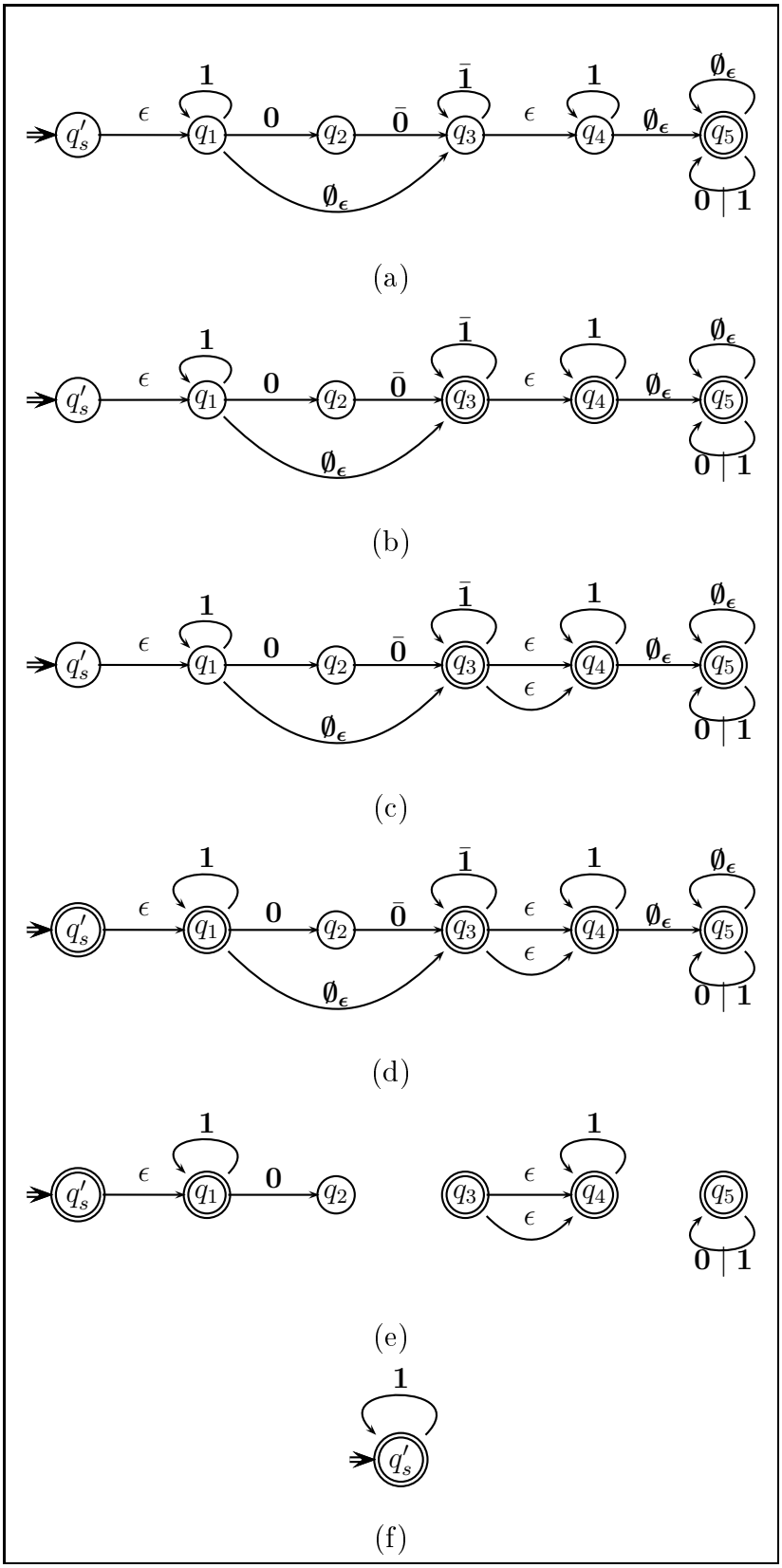


Figure 4.7: Simplification process of automaton corresponding to $\mathcal{L}_{\pi_3}(\texttt{xs})$

automaton. In this automaton all states which lie on the path from the start state to a final state are marked as final. This ensures that the liveness automaton accepts a prefix-

closed language which is needed for the liveness-based garbage collector. The simplified automaton accepts the language $\mathbf{1}^*$ indicating that only the spine of the list `xs` is live. This matches our intuition as only the spine of the list `y` was needed by the **length** function and `y` was constructed by appending `xs` and `ys`.

4.4.1 Liveness-based garbage collection scheme

In this section we describe a garbage collection scheme which uses the result of the liveness analysis described in Section 4.2.1 to collect more garbage. The liveness analysis described in Section 4.2.1 computes program-point-wise stack-liveness and closure-liveness for each closure variable. While we compute liveness at all program points, since the liveness is applied for garbage collection, we need to store liveness only at potential garbage collection points i.e. where the program can potentially allocate heap memory. However, unlike eager languages, where memory from the heap is allocated only for **cons** cells and for passing arguments to functions, in a lazy language each **let** definition requires memory for creating closures. And since such definitions abound in programs, almost every program point in a lazy language can become a trigger point for potential garbage collection, and would therefore require liveness information to be stored. For a practical liveness-based garbage collector, the memory overhead required to store liveness information at all program points may be prohibitively large. Thus, we have to restrict the possible garbage collection points in a function body.

We would like to ensure that a garbage collection is never triggered at a **let** definition inside the body of a function. To do this, we need to ensure that sufficient number of heap cells are available before evaluating a function body. Therefore, we compute the estimated number of heap cells required to create closures while executing the body of the function. For a primitive operation, selector or tester a single heap cell is sufficient. For a function call, assuming that each heap cell can hold a single argument of the function, the number of heap cells is equal to the number of arguments of the called function. Using these estimates for applications we can compute the number of heap cells required for creating closures for a function body. In case the function body has a branch, we consider the maximum among the two branches. During execution, before evaluating a function call, we check whether the available heap cells can cover the estimated number of cells. If the available cells are less than the estimated value a garbage collection is triggered. Although

at the beginning of a function body, the liveness is empty for the current function, the garbage collector can collect garbage from the other functions in the call stack. This way we ensure that before evaluating a function body we always have sufficient cells to create closures for the function body.

While triggering garbage collection at the beginning of a function body suffices when the function body does not have **if** expressions, we need more check points for a function body containing branches. The reason for this is that evaluation of the condition in an **if** expression could potentially lead to evaluation of closures. In case the closure being evaluated is a primitive operator/tester/constructor/selector then no extra space is required as the same cell can be updated to hold the result of evaluation. But if the closure is a function call, evaluation of the function body could lead to creation of more closures requiring extra cells. The calling function does not include these cells as part of its estimate and depends on the called function to check for the required memory. If the condition evaluation in **if** expression triggers the evaluation of a function call, it invalidates the memory check at the beginning of the function body. Therefore, a memory check with the required heap cells for the selected branch with the updated memory availability is necessary. Thus, for each function we store the estimated number of heap cells required at the beginning of the function and at the beginning of each branch. Liveness information is stored at the beginning of each branch[3].

In summary the modified garbage collection scheme is,

1. We statically over-approximate the memory required to create the closures for each function body. On entering a function, if the available memory is less than this requirement for the function, a GC is triggered.
2. Since the evaluation of a **if** condition may trigger a collection, after evaluating the condition the available memory is checked once again against a revised estimate of the memory (based on value of the condition) required to execute the rest of the program. A GC is triggered if enough memory is unavailable.

Another drawback of the analysis is the fact that the closure-liveness is computed once and it remains unchanged. Closure-liveness is computed at the point of creation of the closure by transferring the liveness of the stack variable to which the closure is bound. This liveness is never updated during the analysis even if at a different program point

[3]Storing liveness at beginning of function body is not required as it is empty

the liveness of the associated stack variable itself changes. While this keeps the analysis simple, using constant liveness during garbage collection could leave a lot of garbage uncollected. Considering the liveness of closure variables along only feasible evaluation paths leads to more precise liveness information and improved garbage collection. Assume, for the sake of concreteness, that ep_1 and ep_2 are two evaluation points. During GC, we would like to use more precise liveness, based on the actual paths taken during execution. Therefore, we create separate liveness automata for dependences along paths to ep_1 and ep_2, in addition to automata for dependences along paths to both ep_1 and ep_2. The closure carries the liveness environment for its free variables (as pointers to automata, one for each variable). Initially the liveness environment is based on the dependences along both ep_1 and ep_2. However, after evaluating an **if** condition, the liveness environments are updated to one based on either ep_1 or ep_2, so that subsequent garbage collections are based on more precise liveness information. As an example, in Figure 4.3, a dependence chain for $\mathtt{xs}_{\pi_1}$ begins with the variable `z` at the evaluation point ψ_3 and `z` in turn depends on `y` through $(\textbf{length}\ \mathtt{y}_{\pi_3})$. We denote this chain of dependences as $[\psi_3 : \mathtt{z} \leftarrow (\textbf{length}\ \mathtt{y}_{\pi_3})]$. Indeed, the chains of closures in the heap are runtime representations of these dependences. Since `z` is evaluated at ψ_3 due to the expression **return** `z`, the demand made by the calling context(s) of f places a demand on `z` which will impart a liveness to $\mathtt{y}_{\pi_1}$. Other dependence chains which result in a liveness for $\mathtt{xs}_{\pi_1}$ are $[\psi_1 : \mathtt{c} \leftarrow (\textbf{null?}\ \mathtt{y}_{\pi_4}), \mathtt{y} \leftarrow (\textbf{append}\ \mathtt{xs}_{\pi_3}\ \mathtt{ys}_{\pi_3})]$ and $[\psi_2 : \mathtt{u} \leftarrow (\textbf{car}\ \mathtt{xs}_{\pi_6})]$. The liveness analysis declares the liveness of $\mathtt{xs}_{\pi_3}$ to be a union of the liveness arising from these dependence chains. To be safe, a GC during evaluation of `y` at ψ_1 has to use this liveness to copy the heap starting from $\mathtt{xs}_{\pi_3}$. However, notice that if a GC takes place while evaluating `u` at ψ_3, it can safely consider only the liveness arising from the dependence chain $[\psi_2 : \mathtt{u} \leftarrow (\textbf{car}\ \mathtt{xs}_{\pi_6})]$. The liveness due to the branch terminating with the evaluation point ψ_3 is not feasible after the condition in π_5 evaluates to true.

Figure 4.8 shows the stack and closure liveness for the variable `xs` corresponding to the evaluation point ψ_2. Assuming that the condition `c` evaluates to true, the next expression that is executed has program point π_6. At this point, the stack-liveness of `xs` is only due to `u`. The closure variable $\mathtt{xs}_{\pi_6}$ also gets the same liveness. However, notice that the liveness of the closure variable $\mathtt{xs}_{\pi_3}$, acquired from the variable `y` at π_3 remains unchanged, although there is no future use of `y` in this branch. This leads to less precise

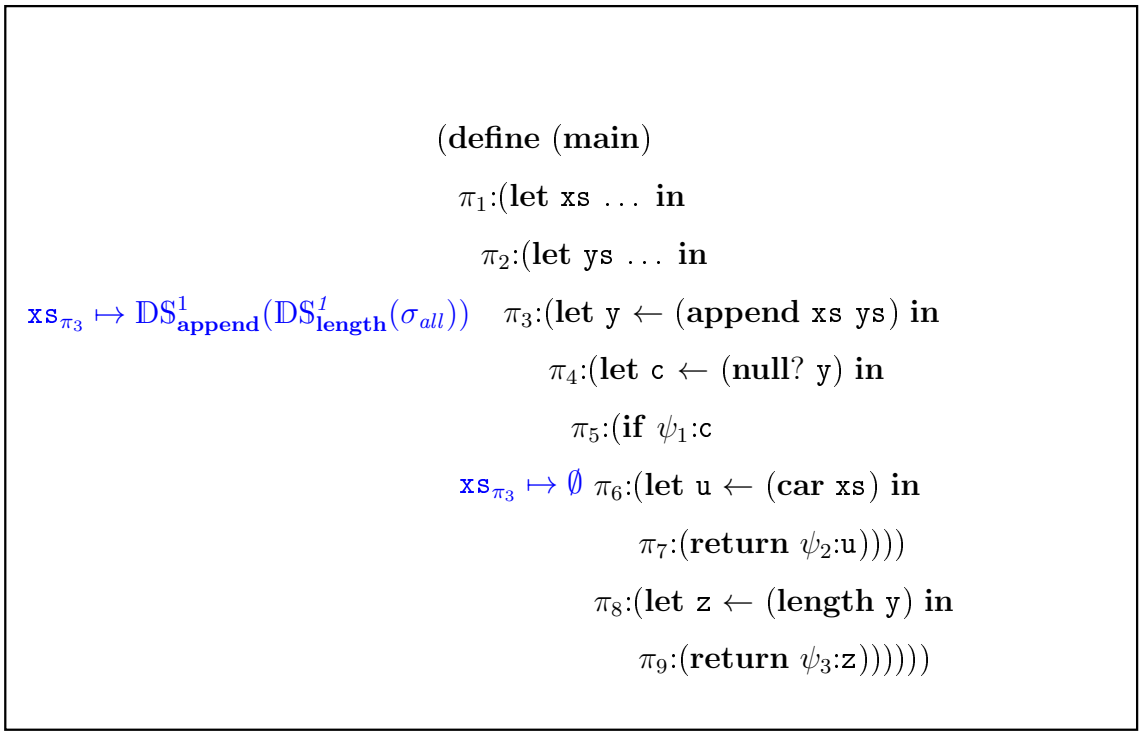


Figure 4.8: Advantages of updating closure liveness for variable `xs` at runtime. Closure liveness of `xs` at π_3 needs to take into account liveness in both branches. Assuming the condition evaluates to true at ψ_1 at runtime, closure liveness of `xs` can be updated to $\emptyset$.

liveness for the closure variable $\texttt{xs}_{\pi_3}$ and may prevent it from being garbage collected. To avoid this, we update the closure-liveness after evaluation of a condition in **if** expression.

4.4.2 The garbage collection algorithm

We shall call a unit of allocatable memory as a *cell*. A cell can hold a basic value (bas), the constructor **cons** (cons arg_1 arg_2) or a closure. The closure, in turn, can be one of (unop arg), (binop arg_1 arg_2) and function application (f arg). Here each arg_i is a reference to another heap cell. In addition, the closure also carries a pointer to a DFA (denoted $\text{arg}_i.\text{dfa}_i$) for each arg_i.

Algorithm 6 describes the garbage collection scheme. Starting with the root set, each cell pointed by a live reference (i.e., whose associated DFA state is final) is copied using copy. Copying a **cons** cell involves copying the cell itself and conditionally copying the **car** and the **cdr** fields after referring to the next states of the DFA. If the reference points to a closure, then, as mentioned earlier, the closure carries pointers to the liveness DFAs of its arguments. These are used to recursively initiate copying of the arguments.

```
procedure lgc():
    for each reference ref in root set:
        ref = copy (ref, init(ref.dfa));
    copyReferencesOnPrintStack();
    return;

function copy(ref, state):
    if ¬final(state):
        return ref;
    newRef = dupHeapCell(ref);
    if ref.cell(ref) is a cons cell (cons arg1 arg2):
        newRef.arg1 = copy(arg1, next(state, 0));
        newRef.arg2 = copy(arg2, next(state, 1));
    if ref.cell is a closure cell, generically (binop arg1 arg2):
        newRef.arg1 = copy(arg1, init(arg1.dfa));
        newRef.arg2 = copy(arg2, init(arg2.dfa));
    return newRef;
```

Algorithm 6: Liveness-based garbage collection.

Note that the copying strategy for (unop arg_1) or (f arg_1) are similar to (binop arg_1 arg_2) and have not been shown.

The evaluation of the top-level expression in a program is driven by a printing function (Section 4.1.1) that is common to all user programs. We describe a generic algorithm for printing values in lazy-languages in Algorithm 7. The print function takes a heap reference as input and checks if it contains an evaluated value or a closure. If it contains a closure, it triggers an evaluation of the closure to produce a value which is in WHNF. In case the value is an atomic value (number) it prints the value. If it is a **cons** cell, then it recursively calls print on the **car** part and then the **cdr** part. Notice that once the **car** part is printed it becomes dead, even though it is reachable from the stack. We extend liveness-based garbage collection to the print function to take advantage of this observation.

```
Function evalAndPrint(ref)
    Data: ref is the expression being evaluated
    val ← evalToWHNF(ref)
    if (pair?(val)):
        Display "("
        evalAndPrint(car(val))
        Display "."
        evalAndPrint(cdr(val))
        Display ")"
    else:
        Display val
```

Algorithm 7: Function to print result of a lazy program.

4.4.3 Experimental evaluation

Our experimental setup consists of the prototypes of (a) an interpreter for our language, (b) a liveness analyzer, and (c) a single generation copying garbage collector. The garbage collector can be configured to work on the basis of reachability (RGC mode) or use liveness DFAs (LGC mode). Our benchmark consists of programs taken from nofib [66] and other repositories for functional programs [2–4]. We ran the experiments on 8 core Intel® Core™ i7-4770 3.40GHz CPU having 8192KB L2 cache, 16GB RAM, running 64 bit Ubuntu 14.04.

The statistics related to liveness analysis and DFA generation are shown in Table 4.1. We observe that the analysis of all programs except `treejoin` and `sudoku` require reasonable time. The bottleneck in our analysis is the NFA to DFA conversion with worst-case exponential behaviour. However, since the analysis has to be done only once and its results can be cached and re-used, the time spent in analysis may be considered acceptable.

Table 4.2 compares GC statistics for RGC and LGC. We report the number of GC events, average number of cells reclaimed per GC, average number of cells touched per GC and the total time to perform all collections. It is no surprise that the number of cells reclaimed per garbage collection is higher and the number of garbage collections lower for LGC. The cost of LGC is higher garbage collection time, which increases the overall execution time even with reduced number of collections. However, the execution time

Table 4.1: Statistics for liveness analysis

Program	#CFG Nonterminals	#CFG Rules	#DFA States	#DFA Transitions	DFA Gen Time (sec)
fibheap	621	1176	1761	2829	37.28
sudoku	1422	2009	4283	7690	655.41
mperm	662	866	1546	2522	0.94
paraffins	1174	1773	3346	6086	13.22
lcss	642	1206	1666	2726	8.66
huffman	499	818	1414	2528	4.00
knightstour	660	883	1519	2420	10.97
nqueens	404	643	889	1170	0.36
deriv	328	468	809	1435	0.61
treejoin	615	1328	1803	2797	903.14
lambda	669	1088	1703	2580	11.01
gc_bench	390	450	571	788	0.10

Data for Liveness Analysis

of LGC is still comparable for most benchmarks (slowdown within 5X of RGC in most cases) and better for 3 benchmarks (2X speedup in the best case).

Memory usage graphs for the benchmarks are shown Figure . In all the programs we can see that the curve corresponding to LGC (blue line) dips below the RGC curve (red line) during GC. The graphs also include the curve for reachable cells (black) and live cells (light-blue). These were obtained by forcing RGC to run at very high frequency. The curve for live cells were obtained by recording heap access times and post processing the data at the end of the program. Note that the size of an LGC cell is 1.16 times the size of a RGC cell.

As demonstrated by the gap between the red and the light-blue lines, a large number of cells which are unused by the program are still copied during RGC. LGC does a much better job of closing this gap but still falls short of the precision achieved by LGC in case of eager languages [12]. A major source of inefficiency in LGC is multiple traversals of already copied heap cells. Since LGC does not mark the heap cells after the first visit, the same cells can be repeatedly visited with different liveness states.

The *huffman* benchmark performs extremely well with liveness-based GC in terms of both the peak memory required and the number of GCs. The benchmark takes a list of characters and first encodes it and then takes the encoded list and decodes it, printing

Table 4.2: Statistics for garbage collection

Program	fibheap	sudoku	nperm	paraffins	lcss	huffman	knightstour	nqueens	deriv	treejoin	lambda	gc_bench
#Cells collected/GC	3466.2	931.3	4684.4	661.7	8064.7	10533.8	179155.0	2607.4	854.6	50284.2	7271.7	14932.7
#Cells collected/GC	4164.5	2328.7	14212.5	2920.5	18268.7	100010.0	312454.0	9529.1	10755.3	936525.0	8448.4	204774.0
#Cells touched/GC	33576.6	3134.6	22743.2	4522.6	14177.9	89536.1	498645.0	7493.4	10269.3	1566250.0	13194.2	189880.0
#Cells touched/GC	50957.9	2950.2	23127.1	2933.0	4604.4	88.6	534562.0	829.0	420.3	700756.0	29156.3	33.2
#GCs	1333	179	710	16	30	356	529	3345	31	116	775	48
#GCs	1108	72	235	4	14	38	304	916	3	5	667	4
Peak Memory Required	37043	4066	27428	5185	22243	100070	677800	10101	11124	1616533	20466	204813
Peak Memory Required	37043	2960	25343	3733	16296	72	642303	1082	589	887005	18169	72
$\frac{LGC}{RGC}$	1.16	0.84	1.07	0.83	0.84	0.00	1.09	0.12	0.06	0.63	1.02	0.00
GC time (sec)	1.46	0.01	0.24	0.00	0.01	0.64	6.24	0.36	0.00	3.84	0.17	0.11
GC time (sec)	20.00	0.04	1.24	0.00	0.02	0.01	64.43	0.18	0.00	1.90	4.70	0.00
$\frac{LGC}{RGC}$	13.70	4.64	5.20	2.01	3.02	0.02	10.32	0.49	0.05	0.49	28.48	0.00
Total Exec time (sec)	13.82	0.11	1.80	0.01	0.13	2.70	55.29	4.68	0.05	6.66	2.77	1.45
Total Exec time (sec)	33.23	0.16	3.39	0.02	0.18	2.55	134.35	5.87	0.05	5.50	8.32	1.22
$\frac{LGC}{RGC}$	2.40	1.5	1.88	1.2	1.36	0.94	2.43	1.25	0.9	0.82	3.00	0.84

Comparing RGC with LGC. Note that the size of an LGC cell is 1.16 times the size of an RGC cell, Total Exec time includes GC time.

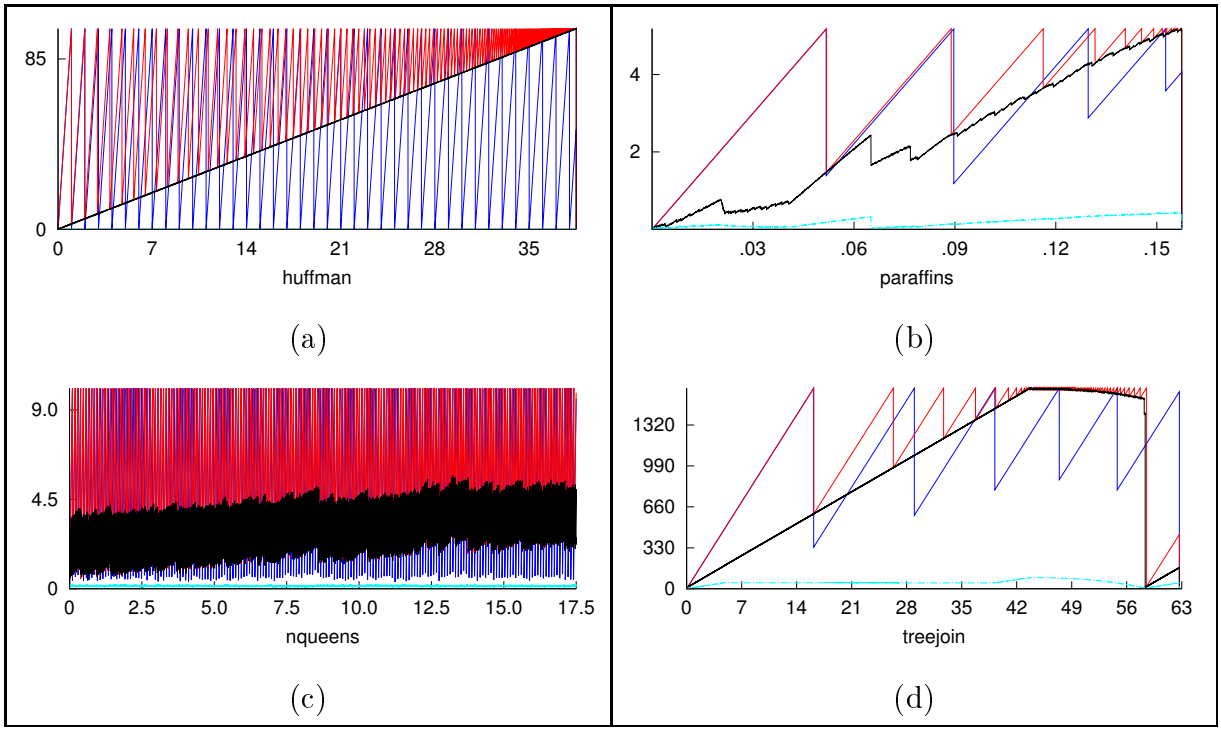


Figure 4.9: Memory usage. The red and the blue curves indicate the number of cons cells in the active semi-space for RGC and LGC respectively. The black curve represents the number of reachable cells and the light-blue curve represents the number of cells that are actually live (of which liveness analysis does a static approximation). x-axis is the time measured in number of cons-cells allocated (scaled down by factor 10^5). y-axis is the number of cons-cells (scaled down by 10^3).

out the decoded list. Notice that in a lazy language, the printing of the final decoded list is what forces the evaluation to move forward. Ideally, once the element is printed the memory allocated to that element can be freed and re-used, but a reachability-based collector will not be able to collect it as a reference to the **cons** cell containing the element will be still on the stack. Since the **evalAndPrint** function is also annotated with liveness information our liveness-based collector will be able to collect it. The input character list to the program is generated using a loop and hence the program can execute in constant memory irrespective of the length of input list. The benchmark demonstrates that our garbage collector is very effective when the program has a producer-consumer nature.

Tail call optimization Tail call optimization is a very important optimization for improving space utilization of programs. In spite of this, many languages do not require

tail calls to be optimized. Not optimizing tail calls not only uses up stack size but it can also hog heap memory if a reachability-based collector is used. For example, a list which is traversed using a tail call will hold references to already processed elements of the list on the stack. This makes them reachable during garbage collection preventing the memory from being garbage collected. Our liveness analysis detects that beyond the tail call the **car** part of the argument list is not used and hence marks it dead. A liveness-based collector would use this information and collect the cells as garbage.

4.5 Soundness of liveness-based garbage collection

We shall now present a proof of the soundness of the liveness-based garbage collection scheme. It is easy to see that the analysis correctly identifies the liveness of stack variables. A stack variable is live between its introduction through a **let** and its last use to create a closure variable. This is correctly captured by the **let** rule. Proving soundness for root set traversals starting with closure variables is more complex. Here are the ideas behind the proof.

1. As in DGS, we extend the abstract machine state $\rho, S, \mathbf{H}, e$ to $\rho, S, \mathbf{H}, e, \delta$. We call such a state a *minefield state*. Here δ is the "dynamic" demand on the expression e. The demand for the initial state is $(\mathbf{0} + \mathbf{1})^*$ (also abbreviated as δ_{all}), and each $\rightsquigarrow$ transition transforms the demand according to the liveness rules of Section 4.2.1. The information in continuation frames on the stack S are also similarly augmented with their demands. Thus, a stacked entry now takes the form (ρ, ℓ, e, δ). The initial state of the minefield semantics is assumed to be $([\,]_\rho, (\rho_{init}, \mathbf{ans}, (\mathbf{evalAndPrint\ ans}), \delta_{all}) : [\,]_S, [\,]_\mathbf{H}, e_\mathbf{main}, \epsilon)$.
2. We augment the standard semantics in Figure 4.2 to simulate a GC before the execution of each **let** definition. We reiterate that, unlike eager languages, memory is allocated only during execution of **let** expressions. $GC(\rho, S, \mathbf{H}, e, \delta)$ models a liveness-based garbage collection that returns $(\rho', S', \mathbf{H}')$. The changes in ρ, S and $\mathbf{H}$ are due to non-live references being replaced by $\perp$[4]. This simulates the act of garbage collecting the cells pointed to by these references during an actual garbage collection. Any attempt

[4]In our fanciful imagination, $\perp$ in the heap are mines and liveness analysis is the mine detector. A wrong liveness analysis can cause $\perp$ to be be dereferenced resulting in the BANG state.

Premise	Transition	Rule name
δ is $\emptyset$	$\rho, (\rho', \ell, e', \delta') : S, \mathbf{H}, e, \delta \rightsquigarrow \rho', S, \mathbf{H}, e', \delta'$	NO-EVAL
	$\rho, (\rho', \ell, e, \delta') : S, \mathbf{H}, \kappa, \delta \rightsquigarrow \rho', S, \mathbf{H}[\ell := \kappa], e, \delta'$	CONST
$\rho(x)$ is $\langle s, \rho' \rangle$	$\rho, S, \mathbf{H}, x, \delta \rightsquigarrow \rho', S, \mathbf{H}, s, \delta$	VAR
$\rho(x)$ is $\bot$	$\rho, S, \mathbf{H}, x, \delta \rightsquigarrow$ BANG	VAR-BANG
$\rho(x)$ is $\langle (\mathbf{id}\ y), \rho' \rangle$	$\rho, S, \mathbf{H}, x, \delta \rightsquigarrow \rho', S, \mathbf{H}, y, \delta$	ID
	$\rho, (\rho', \ell, e, \delta') : S, \mathbf{H}, (\mathbf{cons}\ x\ y), \delta \rightsquigarrow$ $\rho', S, \mathbf{H}[\ell := (\rho(x), \rho(y))], e, \delta'$	CONS
$\mathbf{H}(\rho(x))$ is (v, d)	$\rho, (\rho', \ell, e, \delta') : S, \mathbf{H}, (\mathbf{car}\ x), \delta \rightsquigarrow$ $\rho', S, \mathbf{H}[\ell := v], e, \delta'$	CAR-SELECT
$\mathbf{H}(\rho(x))$ is $\langle s, \rho' \rangle$	$\rho, S, \mathbf{H}, (\mathbf{car}\ x), \delta \rightsquigarrow$ $\rho', (\rho, \rho(x), (\mathbf{car}\ x), \delta) : S, \mathbf{H}, s, \mathbf{0}\delta$	CAR-CLO
$\mathbf{H}(\rho(x))$ is $(\langle s, \rho' \rangle, d)$	$\rho,\ S,\ \mathbf{H},\ (\mathbf{car}\ x), \delta \rightsquigarrow$ $\rho',\ (\rho, addr(\langle s, \rho' \rangle), (\mathbf{car}\ x), \delta) : S,\ \mathbf{H},\ s,\ \delta$	CAR-1-CLO
$\mathbf{H}(\rho(x)), \mathbf{H}(\rho(y)) \in \mathbb{N}$	$\rho, (\rho', \ell, e, \delta') : S, \mathbf{H}, (+\ x\ y), \delta \rightsquigarrow$ $\rho', S, \mathbf{H}[\ell := \mathbf{H}(\rho'(x)) + \mathbf{H}(\rho'(y))], e, \delta'$	PRIM-ADD
$\mathbf{H}(\rho(x)) \notin \mathbb{N}$	$\rho, S, \mathbf{H}, (+\ x\ y), \delta \rightsquigarrow$ $\rho', (\rho, \rho(x), (+\ x\ y), \delta) : S, \mathbf{H}, x, \epsilon$	PRIM-1-CLO
$\mathbf{H}(\rho(y)) \notin \mathbb{N}$	$\rho, S, \mathbf{H}, (+\ x\ y), \delta \rightsquigarrow$ $\rho', (\rho, \rho(y), (+\ x\ y), \delta) : S, \mathbf{H}, y, \epsilon$	PRIM-2-CLO
f defined as $(\mathbf{define}\ (f\ \vec{y})\ e_f)$	$\rho, S, \mathbf{H}, (f\ \vec{x}), \delta \rightsquigarrow$ $[\vec{y} \mapsto \langle (\mathbf{id}\ \vec{x}), \rho \rangle], S, \mathbf{H}, e_f, \delta$	FUNCALL
$GC(\rho_1, S_1, \mathbf{H}_1, (\mathbf{let}\ x \leftarrow s\ \mathbf{in}\ e), \delta)$ $= (\rho, S, \mathbf{H})$, ℓ is a new location	$\rho, S, \mathbf{H}, (\mathbf{let}\ x \leftarrow s\ \mathbf{in}\ e), \delta \rightsquigarrow$ $\rho \oplus [x \mapsto \ell], S, \mathbf{H}[\ell := \langle s, \lfloor \rho \rfloor_{FV(s)}, \delta_x \rangle], e, \delta$ where $\delta_x = \lfloor \mathcal{L}(e, \delta, \mathbb{DS}) \rfloor_{\{x\}}$	LET
$\mathbf{H}(\rho(x)) \neq 0$	$\rho, S, \mathbf{H}, (\mathbf{if}\ x\ e_1\ e_2), \delta \rightsquigarrow \rho, S, \mathbf{H}, e_1, \delta$	IF-TRUE
$\mathbf{H}(\rho(x)) = 0$	$\rho, S, \mathbf{H}, (\mathbf{if}\ x\ e_1\ e_2), \delta \rightsquigarrow \rho, S, \mathbf{H}, e_2, \delta$	IF-FALSE
$\mathbf{H}(\rho(x)) = \langle s, \rho' \rangle$	$\rho, S, \mathbf{H}, (\mathbf{if}\ x\ e_1\ e_2), \delta \rightsquigarrow$ $\rho', (\rho, \rho(x), (\mathbf{if}\ x\ e_1\ e_2), \delta) : S, \mathbf{H}, x, \epsilon$	IF-CLO
$\mathbf{H}(\rho(x)) = \langle s, \rho' \rangle$	$\rho, S, \mathbf{H}, (\mathbf{return}\ x), \delta \rightsquigarrow$ $\rho',\ (\rho, \rho(x), (\mathbf{return}\ x), \delta) : S, \mathbf{H}, x, \delta$	RETURN

Figure 4.10: Minefield semantics. The differences with the small-step semantics have been highlighted by shading.

to dereference such references during execution results in the transition system entering a special state denoted BANG. $GC(\ldots)$ needs to consider the following environments: (1) the environment in the current state, (2) the environment in each of the stacked continuations and (3) the environment in each of the closures in the heap.

   (a) For each of these environments, $GC(\ldots)$ calculates a liveness environment $\mathcal{L}$ for the corresponding s with the dynamic demand δ.

   (b) For each location ℓ, $GC(\ldots)$ sets $\mathbf{H}(\ell)$ to $\perp$ iff for each environment ρ above, for each $x \in \mathit{domain}(\rho)$, and each prefix α' of an access path α, it is *not* the case that $x.\alpha \in \mathcal{L}$ and $\mathbf{H}[\![\rho(x), \alpha']\!] = \ell$.

3. Note that the modeled GC is done on the basis of dynamic demand rather than static demand. However, by a reasoning similar to Theorem 3.4, the static liveness that is consulted during actual GC is computed from an over-approximation of this demand. Thus, the soundness result on the modeled GC will also apply for the actual GC. The soundness proof consists in showing that the execution of no program enters the BANG state.

Figure 4.10 shows the minefield rules. As mentioned earlier, the transition for a **let** is preceded by $GC(\ldots)$. Also consider, as an example, the transition for the modified CAR-CLO. If an earlier call to $GC(\ldots)$ results in $\rho(x)$ being bound to $\perp$, then the $\rightsquigarrow$ step enters the BANG state (CAR-BANG). Otherwise, the transition is similar to the earlier CAR-CLO rule.

4.5.1 Soundness result

Note that our proofs will be for a single round of minefield execution i.e., the evaluation of (**main**) to its WHNF driven by the printing mechanism (Section 4.1.1). With minor variations, the proof will also be applicable for subsequent evaluations initiated by **evalAndPrint**. We now present the result which shows that the liveness-based garbage collection scheme is sound.

Theorem 4.1 *The minefield execution of no program can enter a* BANG *state.*

PROOF. Consider a state $(\rho, S, \mathbf{H}, e, \delta)$ in the minefield execution of a program. We show by induction on the number n of $\rightsquigarrow$ steps leading to this state that the next transition cannot enter a BANG state. When n is 0, the state is $([\,]_\rho,\ (\rho_{init},\ \ell_{\mathbf{ans}},\ (\mathbf{evalAndPrint\ ans}), \delta_{all})$:

$[\,]_S, [\,]_{\mathbf{H}}, e_{\mathbf{main}}, \epsilon)$. Since the call to $GC(\ldots)$ in this state does nothing, we just have to show that the $\rightsquigarrow$ transition cannot enter a BANG state. Since our programs are in ANF, $e_{\mathbf{main}}$ can only be a **let** expression. A LET step does not involve dereferencing, and thus cannot result in a BANG.

For the inductive step, we shall show that none of the minefield steps that involves dereferencing results in a BANG. These are the steps which have a $\mathbf{H}(\rho(...))$ in the premise. If we are in the the state $(\_, \_, \_, e, \delta)$ after n steps of minefield execution, then, because of the NO-EVAL rule δ is not null. Now a $\rightsquigarrow$ step can go BANG because it dereferences a $\perp$ inserted by an earlier $GC(\ldots)$. However, again by a reasoning similar to Lemma 3.2, the demand δ' on the basis of which the $GC(\ldots)$ would have inserted a $\perp$ would have included the current demand δ. Thus it is enough to show that the $\rightsquigarrow$ step would also not lead to a BANG.

We only consider the rules for the case when e is (**car** `x`). The rest of the rules involve similar reasoning. For the CAR-CLO rule in the state $(\rho, S, \mathbf{H}, (\mathbf{car}\ x), \delta)$, we know that δ is non-null. Therefore the liveness of x includes ϵ. Now since garbage collection on (**car** x) was done on the basis of a demand that included δ, $GC(\ldots)$ would not have inserted a $\perp$ for x and therefore the dereferencing $\mathbf{H}(\rho(x))$ will go without BANG.

Similarly for the CAR-1-CLO rule, observe that there are two dereferences. First x is dereferenced to get a cons cell and then the head of the cons cell is dereferenced to obtain a closure. If the demand δ on (**car** x) is non-null, then the liveness of x will have as its prefixes both ϵ and $\mathbf{0}$, and $GC(\ldots)$ on (**car** x) with a liveness that includes δ will neither bind x to a $\perp$, nor insert $\perp$ at the first component of the cons cell. Thus both dereferences can take place without $\rightsquigarrow$ entering the BANG state. ■

4.6 Related work

The impact of liveness on the effectiveness of GC is investigated in [35]. They observe that liveness can significantly impact garbage collection, but only when it is interprocedural. As far as memory requirement is concerned, our paper demonstrates this observation. To the best of our knowledge, this is the first work that uses the results of an interprocedural liveness analysis to garbage collect both evaluated data and closures. Thomas [96] describes a copying garbage collector for the Three Instruction Machine (TIM) [25] that

only preserves live closures in a function's environment (also called a frame). However, in the absence of details, it is not clear whether a) the scope of the method is interprocedural, and b) it handles algebraic datatypes like lists (the original design of TIM did not). All other previous attempts [12, 45, 87–89] involved either imperative or eager functional languages.

There have been several attempts to use liveness analysis to improve GC for imperative languages. [49] presents a liveness analysis and uses the results for inserting nullifying statements in Java programs. In [89], temporal properties like liveness are checked against an automaton modeling heap accesses. Both these approaches are intraprocedural in scope.

In the space of functional languages, there are: rewriting methods such as deforestation [21, 29, 99], sharing analysis based reallocation [75], region based analysis [98], and insertion of compile-time nullifying statements [41, 56]. The analysis described by Inoue et. al. [41] handles the specific case that arises when list-valued functions F and G are used in an expression of type $F(G(\ldots))$. If a cell c created by G and is not part of the result returned by F, c can be garbage collected whenever F completes execution. Similar to our method, the result of their analysis is also represented using grammars(CFG). However their method introduces (under)approximation in the CFG itself to remove symbols equivalent to $\bar{\mathbf{0}}$ and $\bar{\mathbf{1}}$ from CFG rules. Another approach to detect garbage cells generated by expressions of type $F(G(\ldots))$ due to Mohnen [62] uses abstract interpretation. A list having n levels is abstracted to an n-tuple, a boolean denoting the possibility of sharing between any element at each level in the list and the result of the function to which the list is passed as a parameter. A false value in the tuple indicates that values at that level are not shared with the return value and hence can be garbage collected. This leads to very coarse approximations as the use of a single cell will make the whole list at that level live. The bigger limitation with both approaches is that garbage collection can happen only at end of function bodies.

Another approach due to Lee et. al. [55, 56] uses *memory types* to describe usage of heap cells by expressions. Their analysis also achieves context sensitivity by doing a parameterized analysis of functions. The method uses dynamic flags passed as extra arguments to functions to collect cells inside function bodies. Passing different values from different call sites for the dynamic flags allows the same function to have different

deallocation behaviors.

Another method that comes close to our approach is the *Heap Safety Automaton* [89]. The goal of this method is to safely insert null statements and it uses an automaton to model safety of null insertion statement at a given point. The program is abstracted by a shape graph and this graph is used to do model-checking against the heap safety automaton. A key aspect of any approach which tries to improve precision of garbage collection is to identify the earliest program point where a reference can be set to null. The Heap Safety Automaton based approach does not address this issue, it can only answer whether a given access expression be set to null immediately after a program point. It fails to answer the following question, Which expressions should be considered at which program point? This is a very crucial question as considering every pair of access expression and program point is impractical.

A practical approach involves copying only the heap objects whose root variables are live [6]. The drawback of this approach is that an entire object reachable from a live root variable is considered live, even if some parts of it are never used. For example, even when only the spine of a list is live (used as an argument to the **length** function) all its elements will also be copied.

The only work in the space of lazy languages seems to be [31] which touches upon only basic techniques of compile-time garbage marking, explicit deallocation and destructive allocation. An interesting approach suggested in [36] is to annotate the heap usage of first-order programs through linear types. The annotations are then used to serve memory requests through re-allocation. However, this requires the user to write programs in a specific way. Safe-for-space [11] approaches [23, 90] reduce the amount of heap used by a program by allocating closures in registers and through tail call optimizations. However, these approaches take care of only part of the problem addressed by our analysis as the program can still contain unused objects and closures that are reachable.

Simplifiers [68] are abstractly described as lightweight daemons that attach themselves to program data and, when activated, improve the efficiency of the program. Our liveness-based GC can be seen as an instance of a simplifier which is tightly coupled with garbage collectors. The approach that is closest to the method described in this paper is the liveness-based garbage collector implemented in [12, 44] and address eager languages. We extend this to handle lazy evaluation and closures.

Chapter 5

Static program slicing using demand propagation

Program slicing is a powerful technique that is widely used for debugging, software maintenance, optimization, program analysis and information flow control. Program slicing refers to the class of techniques that delete parts of a given program while preserving certain desired behaviours. The desired behaviors are specified using what is called as the 'slicing criterion'. According to the original definition of slicing given by Weiser [103], a slice of a program $\mathbb{P}$ with respect to a statement `x` and variable `v` is the set of statements of $\mathbb{P}$ which affect the value of `v` at statement `x` for all possible inputs. The main applications of slicing include software testing [15, 30, 32, 33, 38], program debugging [58, 102], measurement [14, 69–71], validation [50], program parallelization [103], program integration [37], reverse engineering and program comprehension [1, 13], program restructuring [18, 22, 43, 51, 94], program specialization [79] and identification of reusable functions [10, 19, 54].

The definition of what constitutes a slice has been modified in multiple ways depending on the application. We consider one such version of slicing where the slicing criterion identifies parts of the final output of the program and the goal is to produce only those parts of the program which affect the parts of the output identified by the slicing criterion. Program specialization, parallelization, dead code analysis and cohesion measurement are examples of such applications. In this chapter, we formulate the slicing problem as a dependence analysis problem and use the analysis defined in Chapter 3.2 to solve it. As an interesting consequence of our formulation, we were able to come up with a novel and

efficient way of slicing called *Incremental slicing* to slice the same program multiple times with different criteria. The soundness of our slicing algorithm follows directly from the soundness of dependence analysis. We prove the correctness of incremental slicing with respect to the non-incremental slicing method.

5.1 Program slicing using dependence analysis

The example from [79] shown in Figure 5.1a motivates the need for program slicing. It takes a string as input and returns a pair consisting of the number of characters and lines in the string. Figure 5.1b shows the program when it is sliced with respect to the first component of the output pair, namely the number of lines in the string (`lc`). All references to the count of characters (`cc`) and the expressions responsible for computing `cc` only have been sliced away (denoted □). The same program can also be sliced to produce only the char count and the resulting program is shown in Figure 5.1c. ■

Formally, Weiser [103] defines slicing criterion as a pair $\langle p, V \rangle$, where p is a program point and V is a subset of program variables. A program slice on the slicing criterion $\langle p, V \rangle$ is a subset of program statements that preserves the behavior of the original program at the program point p with respect to the program variables in V, i.e., the values of the variables in V at program point p are the same in both the original program and the slice. Similarly, we define slicing criterion for a functional program P as the pair $\langle e, \sigma \rangle$, where e represents a particular expression in the program $\mathbb{P}$ and σ represents the parts of the value of e that is of interest. The goal of slicing is to identify the set of expressions belonging to $\mathbb{P}$ which may affect the parts identified by σ. In general, the question we want to answer is: Given a slicing criterion $\langle e, \sigma \rangle$, which other expressions $e_i \in \mathbb{P}$ decide σ part of the value of e?

Notice the similarity between the slicing problem and the problem of computing dependences in functional programs. Indeed, given a slicing criterion $\langle e, \sigma \rangle$, an expression e_i decides the value of σ part of the value of e only if σ part of e depends on e_i. We view the slicing criterion (a set of strings over $(\mathbf{0} + \mathbf{1})^*$) as a demand on the expression (**main**), using this we compute the demand on each expression in the program. Unlike liveness analysis where the demand is always $(\mathbf{0} + \mathbf{1})^*$, the slicing criterion can be any subset of $(\mathbf{0} + \mathbf{1})^*$ and is supplied by a context that is external to the program. To decide

```
(define (lcc str lc cc)
    (if (null? str)
        (return (cons lc cc))
        (if (eq? (car str) nl)
            (return (lcc (cdr str) (+ lc 1) (+ cc 1)))
            (return (lcc (cdr str) π1:lc π2:(+ cc 1))))))
(define (main)
    (return (lcc ... 0 0))))
(main)
```

(a) Program to compute the number of lines and characters in a string.

```
(define (lcc str lc □)
    (if (null? str)
        (return (cons lc □))
        (if (eq? (car str) nl)
            (return (lcc (cdr str) (+ lc 1) □))
            (return (lcc (cdr str) π1:lc π2:□)))))
(define (main)
    (return (lcc ... 0 □))))
(main)
```

(b) Slice of program in (a) to compute the number of lines only

```
(define (lcc str □ cc)
    (if (null? str)
        (return (cons □ cc))
        (if (eq? (car str) nl)
            (return (lcc (cdr str) □ (+ cc 1)))
            (return (lcc (cdr str) π1:□ π2:(+ cc 1))))))
(define (main)
    (return (lcc ... □ 0))))
(main)
```

(c) Slice of program in (a) to compute the number of characters only.

Figure 5.1: A program in Scheme-like language and its slices. The parts that are sliced away are denoted by □.

```
1.  (define (mmp xs p nv np xv xp)
2.    (if (null? xs)
3.        (return (cons (cons nv np) (cons xv xp)))
4.        (let  p1← (+ π:p 1) in
5.            (if (< (car xs) nv)
6.                (mmp (cdr xs) p1 (car xs) p xv xp))
7.                (if (> (car xs) xv)
8.                    (mmp (cdr xs) p1 nv np (car xs) p)
9.                    (mmp (cdr xs) p1 nv np xv xp)))))

10. (define (main)
11.   (return (mmp (cdr xs) 2 (car xs) 1 (car xs) 1)))
12. (main)
```

Figure 5.2: A program to compute the min and max elements in a list along with their positions.

whether to slice a particular expression e_i, we only need to know whether the part of the program output identified by the slicing criterion is dependent on e_i or not. Specifically, if the demand on the expression e_i turns out to be $\emptyset$, the expression does not contribute to the parts identified by the slicing criterion and can be removed from the slice. We now formally define the slicing problem and show that it is undecidable.

Definition 5.1 *The slicing problem is to find an algorithm $\mathbb{A}$ such that given a program P, a demand δ, and a control point π will answer* yes *if there exists a DGS trace of P in which π occurs with a non-null demand, and* no *otherwise.*

Theorem 5.2 *The slicing problem as stated in 5.1 is undecidable.*

PROOF. To prove this, it is enough to prove that the problem of whether for an arbitrary grammar $G \in CG'$ (described in Section 2.4.1) the language $\mathscr{L}(G) = \emptyset$ is undecidable.

Assume to the contrary that we have a null-checker that can decide whether $\mathscr{L}(G)$ is empty or not. We show that this implies that the ϵ-recognition problem of G also becomes decidable, thereby resulting in a contradiction. If the null-checker for $\mathscr{L}(G)$ says yes, then it is clear that $\epsilon \notin \mathscr{L}(G)$. Otherwise consider a string $\alpha \in \mathscr{L}(G)$. Clearly,

$\mathsf{S} \stackrel{*}{\Rightarrow} \alpha$ and we can assume without loss of generality that in this derivation S-productions appear before any other category of productions. Thus, the derivation must go through a sentential form containing S^c_{final} for the first time, say $L\alpha_l\mathsf{S}^c_{final}\alpha_r R$. In the segment of the derivation from $L\mathsf{S}_{init}wR$ to $L\alpha_l\mathsf{S}^c_{final}\alpha_r R$ consider any production. This production will correspond to a valid TM move. Thus the TM will reach a halting state. However, by Lemma 2.7, it follows that, starting from S, there is an (possibly) alternate derivation $\mathsf{S} \stackrel{*}{\Rightarrow} \epsilon$. Thus, either $\mathscr{L}(G) = \emptyset$ or $\epsilon \in \mathscr{L}(G)$. So, if the null-checker returns `no`, we can conclude that $\epsilon \in \mathscr{L}(G)$. This gives an algorithm for deciding the ϵ-membership of G, a contradiction. Hence the emptiness question of G is undecidable. ■

It turns out that similar to the ϵ-membership question, the emptiness question also becomes decidable if the grammar is regular. Therefore we model our slicing problem as a dependence analysis problem. Modelling the slicing problem as an instance of dependence analysis in Chapter 2 provides several advantages: i) the previously introduced notion of access paths (set of prefix-closed strings of selectors) is rich enough to define interesting and meaningful slicing criteria, ii) the slicing method shares the advantages of computing context independent summaries for user-defined functions that can be used to analyze function call without analyzing the function body multiple times, and most importantly, iii) function summaries allow us to define a very efficient, incremental way of slicing a program multiple times with different criteria. The incremental slicing algorithm will be discussed in Section 5.2.

5.1.1 Slicing algorithm

For the rest of the discussion, we consider the program in Figure 5.2 as our running example. The program takes a list of integers as input and computes the minimum and maximum values along with their positions in the input list. The function **mmp** keeps track of the current minimum and maximum value using the arguments `xv` and `nv`. It compares every element with `xv` and if the current element is greater, it updates `xv` to be the current element and processes the rest of the list. `p` which keeps track of the position of the current element is used to update `xp`. The minimum value and its position are also computed similarly.

We can extract different slices from the example program by specifying different

Function computeSlice(P, σ)
Data: program P, slicing criteria σ
Result: Slice of P for the slicing criterion σ
$S \leftarrow P$
$DE \leftarrow$ Compute demand environment for P with demand on **main** as σ
foreach *(grammar* $G_\pi \in DE$*)***do**
$M_\pi \leftarrow$ Over-approximate G_π using Mohri-Nederhof transformation
$M'_\pi = \mathcal{S}(M_\pi)$
Mark as final each state in M'_π which has a path to a final state in M'_π
foreach *(*$\pi : e \in S$*)***do**
if *(*$\mathscr{L}(M'_\pi) = \emptyset$*)*:
Replace $\pi : e$ with $\square$
return S

Algorithm 8: Function to compute slice of a program using dependence analysis.

slicing criterion. For example, we might be interested in only the maximum value and its position in the list, or we might be interested in only the maximum and minimum values without needing their positions. The demand $\{\epsilon, \mathbf{1}, \mathbf{10}, \mathbf{01}\}$ selects the part of the output which corresponds to the maximum value and its position and similarly, the demand $\{\epsilon, \mathbf{0}, \mathbf{1}, \mathbf{10}, \mathbf{00}\}$ selects only the maximum and minimum values. We compute the demand environment for the example program with the given slicing criterion σ as the demand on (**main**). The question whether the expression at π can be sliced for the slicing criterion σ is equivalent to asking whether the language $\mathcal{S}(\mathscr{L}(\mathsf{D}_\pi))$ is empty.

Algorithm 8 describes our slicing algorithm. It takes a program $\mathbb{P}$ and slicing criterion σ as input. It computes the demand environment for $\mathbb{P}$ using σ as the demand on (**main**) of $\mathbb{P}$. We use Mohri-Nederhof transformation to over-approximate any context-free grammars by strongly regular grammars. The strongly-regular grammars are then converted to a set of non-deterministic finite automata (NFA) and the simplification operation $\mathcal{S}()$ is performed on these NFA. Post simplification, we ensure that the language generated is prefix-closed by setting all states that are in a path from the start node to a final node as final (including the start state). Finally, the required slice is computed by checking if the language generated by the grammar corresponding to an expression is empty and removing the expression from the slice if language is empty. We now use our running

example to explain our slicing algorithm. We show the working of our slicing algorithm for the slicing criteria $\{\epsilon, \mathbf{1}, \mathbf{10}, \mathbf{01}\}$ and $\{\epsilon, \mathbf{0}, \mathbf{1}, \mathbf{10}, \mathbf{00}\}$. Specifically, we consider the occurrence of the variable p identified by the program point π at line 4 and check whether it can be sliced or not for the given slicing criterion. Consider $\mathbb{DS}^2_{\mathbf{mmp}}$, the function that propagates the demand on a call to **mmp** to its second argument. Firstly notice that, according to the rules of **if** and **let**, the demand σ is propagated without change to the three calls at lines 6, 8 and 9. Further, p appears as the fourth argument to the call to **mmp** at line 6 and the sixth in the call to **mmp** at line 8. Clearly the demands on these two occurrences of p are $\mathbb{DS}^4_{\mathbf{mmp}}(\sigma)$ and $\mathbb{DS}^6_{\mathbf{mmp}}(\sigma)$. Also notice that the demands of the three occurrences of p1 at lines 6, 8 and 9 are the same, namely $\mathbb{DS}^2_{\mathbf{mmp}}(\sigma)$. And since p is being used to define p at line 4, by the **let** rule, the demand on this occurrence of p is:

$$\text{if } (\mathbb{DS}^2_{\mathbf{mmp}}(\sigma)) \neq \emptyset \text{ then } \{\epsilon\} \text{ else } \emptyset$$

Bringing everything together, we get:

$$\begin{aligned} \mathbb{DS}^2_{\mathbf{mmp}}(\sigma) &= \mathbb{DS}^4_{\mathbf{mmp}}(\sigma) \cup \mathbb{DS}^6_{\mathbf{mmp}}(\sigma) \cup \\ &\qquad \text{if } (\mathbb{DS}^2_{\mathbf{mmp}}(\sigma)) \neq \emptyset \text{ then } \{\epsilon\} \text{ else } \emptyset \end{aligned}$$

We have to bring this equation to a closed-form by substituting the values of $\mathbb{DS}^4_{\mathbf{mmp}}(\sigma)$ and $\mathbb{DS}^6_{\mathbf{mmp}}(\sigma)$ and eliminating the recursion in $\mathbb{DS}^2_{\mathbf{mmp}}(\sigma)$.

The equations shown below define $\mathbb{DS}^2_{\mathbf{mmp}}(\sigma)$. Notice that the earlier equation for $\mathbb{DS}^2_{\mathbf{mmp}}(\sigma)$ has been rewritten in terms of the symbols $\bar{\mathbf{0}}$, $\bar{\mathbf{1}}$ and $\emptyset_{\epsilon}$.

$$\begin{aligned} \mathbb{DS}^2_{\mathbf{mmp}}(\sigma) &= \mathbb{DS}^6_{\mathbf{mmp}}(\sigma) \cup \mathbb{DS}^4_{\mathbf{mmp}}(\sigma) \cup \emptyset_{\epsilon}\mathbb{DS}^2_{\mathbf{mmp}}(\sigma) \\ \mathbb{DS}^4_{\mathbf{mmp}}(\sigma) &= \bar{\mathbf{1}}\bar{\mathbf{0}}\sigma \cup \mathbb{DS}^4_{\mathbf{mmp}}(\sigma) \\ \mathbb{DS}^6_{\mathbf{mmp}}(\sigma) &= \bar{\mathbf{1}}\bar{\mathbf{1}}\sigma \cup \mathbb{DS}^6_{\mathbf{mmp}}(\sigma) \end{aligned}$$

Assuming the concrete demand on the body of **mmp** to be $\sigma_{\mathbf{mmp}}$, it is easy to see that this demand propagates without change to the three calls in the body of **mmp**. The call in **main** has a demand that is the same as the slicing criterion σ_{sc}. Thus we get:

$$\sigma_{\mathbf{mmp}} = \sigma_{\mathbf{mmp}} \cup \sigma_{sc}$$

Which gives the value of $\sigma_{\mathbf{mmp}}$ as σ_{sc}. When the body of **mmp** is analyzed with the demand $\sigma_{\mathbf{mmp}}$, the demand on p1 is $\mathsf{DS}^2_{\mathbf{mmp}}\sigma_{\mathbf{mmp}}$. Thus, by the let rule, the demand on p at π, denoted D_{π}, is $\emptyset_{\epsilon}\mathsf{DS}^2_{\mathbf{mmp}}\sigma_{\mathbf{mmp}}$.

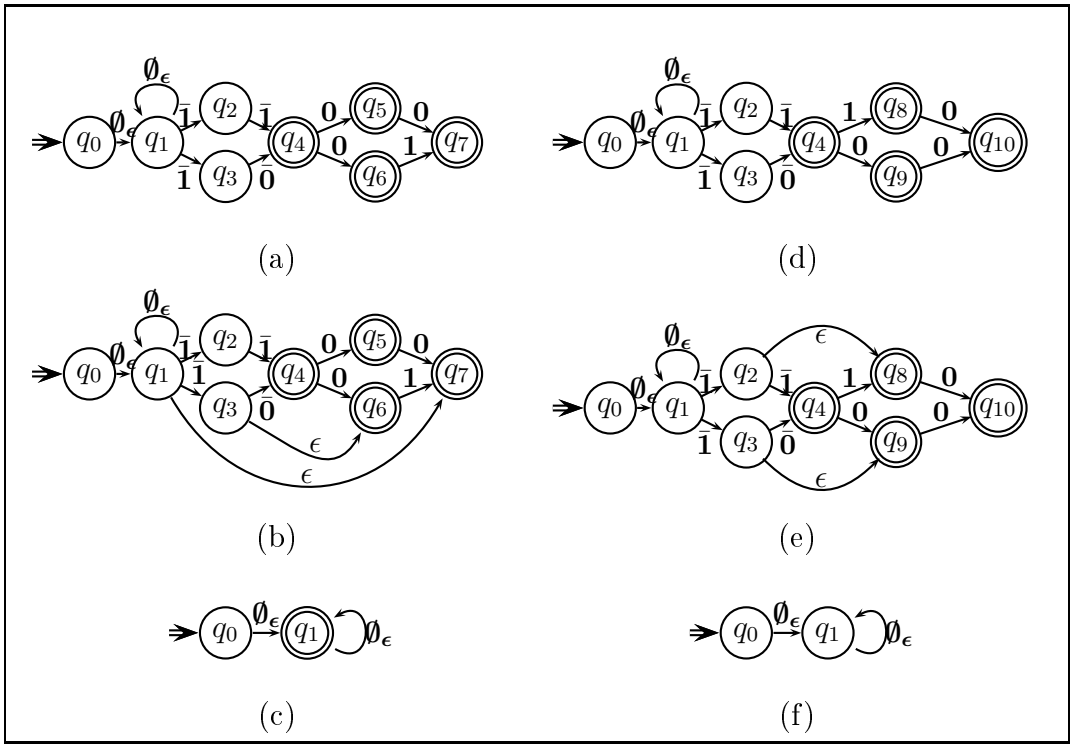


Figure 5.3: The simplification of the automaton M_π^σ: (a), (b) and (c) show the simplification for the slicing criterion $\sigma = \{\epsilon, \mathbf{0}, \mathbf{01}, \mathbf{00}\}$, while (d), (e) and (f) show the simplification for the criterion $\sigma = \{\epsilon, \mathbf{1}, \mathbf{0}, \mathbf{10}, \mathbf{00}\}$.

The equations can now be re-written as: grammar rules:

$$
\begin{aligned}
\mathsf{D}_\pi &\to \boldsymbol{\emptyset_\epsilon}\mathsf{DS}^2_{\mathbf{mmp}}\sigma_{\mathbf{mmp}} \\
\mathsf{DS}^2_{\mathbf{mmp}} &\to \mathsf{DS}^4_{\mathbf{mmp}} \mid \mathsf{DS}^6_{\mathbf{mmp}} \mid \boldsymbol{\emptyset_\epsilon}\ \mathsf{DS}^2_{\mathbf{mmp}} \\
\mathsf{DS}^4_{\mathbf{mmp}} &\to \bar{\mathbf{1}}\bar{\mathbf{0}} \mid \mathsf{DS}^4_{\mathbf{mmp}} \\
\mathsf{DS}^6_{\mathbf{mmp}} &\to \bar{\mathbf{1}}\bar{\mathbf{1}} \mid \mathsf{DS}^6_{\mathbf{mmp}} \\
\sigma_{\mathbf{mmp}} &\to \sigma_{\mathbf{sc}}
\end{aligned}
$$

Similar to liveness, we are interested in the least solution of equations as it corresponds to the most precise slice. For our running example, the grammar after dependence analysis is already regular, and thus remains unchanged by Mohri-Nederhof transformation. The automata in Figure 5.3a–c and 5.3d–f correspond to the two slicing criteria $\sigma_{\mathbf{mmp}} = \{\epsilon, \mathbf{0}, \mathbf{00}, \mathbf{01}\}$ and $\sigma_{\mathbf{mmp}} = \{\epsilon, \mathbf{0}, \mathbf{00}, \mathbf{1}, \mathbf{10}\}$ and illustrate the simplification of corresponding Mohri-Nederhof automata $M_\pi^{\sigma_{\mathbf{mmp}}}$. It can be seen that, when the slicing criterion is $\{\epsilon, \mathbf{0}, \mathbf{00}, \mathbf{1}, \mathbf{10}\}$, the language of D_π is empty and hence the argument `p` can be sliced away, giving us the required slice.

In our formulation, any expression which gets a demand $\emptyset$ for a given slicing criterion is considered dead code and can be removed from the program. As a consequence of this, the dead code elimination compiler optimization becomes a special case where the slicing criterion is set to $(\mathbf{0}+\mathbf{1})^*$.

The correctness of our static slicing algorithm follows from the correctness of our dependence analysis. For a given slicing criterion σ, our slicing algorithm replaces expressions with $\Box$ only when the statically computed demand on it is $\emptyset$. This implies that no DGS trace of the program with demand on **main** as σ will evaluate the removed expression.

Lemma 5.3 *The static slicing algorithm described in Algorithm 8 is sound.*

5.2 Incremental Slicing

Applications such as program specialization, cohesion measurement and parallelization require the same program to be sliced with more than one slicing criterion. These applications can benefit from an incremental static slicing method in which some of the computations for slicing with respect to one criterion could be reused for another.

In this section, we consider the problem of incremental slicing for first order functional programs. The incremental algorithm avoids the repetition of computation when the same program is sliced with different criteria. This is done by a one time precomputation that computes that part of the slice which is common to all slicing criteria. The result is then converted to a set of automata, one for each expression in the program. This completes the precomputation step. To decide whether an expression is in the slice for a given slicing criterion, we convert the slicing criterion to a NFA and check if the intersection of this NFA with the precomputed automaton is $\emptyset$. If the result is $\emptyset$, the expression can be sliced out.

The reason for the efficiency of our incremental method is that most of the efforts in slicing can be factored out in a one time precomputation step (per program) which computes, for each expression, *all* slicing criteria that keep an expression in the slice and store it as an NFAs. Even more interesting, we can compute all the slicing criteria that keep an expression in the slice by performing a dependence analysis on the program with $\sigma_{\mathbf{main}} = \epsilon$. The NFAs which result from the dependence analysis are converted

into a canonical form which we will shortly discuss. The purpose of this conversion is to ensure that the language generated by the NFAs have $(\bar{\mathbf{0}} + \bar{\mathbf{1}})$s only towards the end of the string. Now, given such an NFA, we can construct a corresponding NFA, called *completing automaton*, which exactly captures strings that would cancel out the bar-edges in the original NFA. The completing automata can be stored and slicing with a specific criterion is a small additional computation over the result of this precomputation step.

5.2.1 Motivating example

We will use the example in Figure 5.4 to motivate the need for an incremental slicing algorithm. Recall that the program takes a list of integers as input and computes the minimum and maximum values along with their positions in the input list. The function **mmp** keeps track of the current minimum and maximum value using the arguments `xv` and `nv`. The original program can be specialized to compute only the min and max values in the list (Figure 5.4b) by slicing the program with the slicing criterion $\{\epsilon, \mathbf{0}, \mathbf{1}, \mathbf{00}, \mathbf{10}\}$. Similarly, by using the slicing criterion $\{\epsilon, \mathbf{0}, \mathbf{00}, \mathbf{01}\}$, we get a program which computes the minimum value and its position (Figure 5.4c). If we were to use the slicing algorithm described in Algorithm 8, the process of computing automata, simplification of automata has to be repeated. The goal of our incremental algorithm is to avoid the duplication of this effort.

The key observation driving our incremental slicing algorithm is the fact that the we treat the slicing criterion to be prefix-closed. Therefore, a certain kind of containment relation exists between the slices of a program. We say that a slice P contains Q, if all the expressions in Q are also present in P. For example, the slice corresponding to the criterion $\{\epsilon, \mathbf{0}\}$ should contain the slice corresponding to the criterion $\{\epsilon\}$. Since $\{\epsilon\}$ criterion is the smallest non-trivial slicing criterion that can be used, an expression that is sliced away in the ϵ-slice cannot be part of a slice computed using any other slicing criterion. We use this observation to perform a precomputation step to slice the original program with ϵ slicing criterion. The result of this slicing is then processed to bring it to a certain canonical form and stored. Given a slicing criterion the stored results are used to compute the slice corresponding to the given slicing criterion.

We now describe the actual steps of our precomputation in detail:

1. Use the non-incremental slicing method with slicing criterion $\{\epsilon\}$ to compute the de-

```
(define (mmp xs p nv np xv xp)
   (if (null? xs)
       (return (cons (cons nv np) (cons xv xp)))
       (let  p1← (+ π : p 1) in
           (if (< (car xs) nv)
               (mmp (cdr xs) p1 (car xs) p xv xp))
               (if (> (car xs) xv)
                   (mmp (cdr xs) p1 nv np (car xs) p)
                   (mmp (cdr xs) p1 nv np xv xp)))))
  (define (main)
   (return (mmp (cdr xs) 2 (car xs) 1 (car xs) 1)))
  (main)
```

(a) Program to compute the min and max values in the list

```
(define (mmp xs □ nv □ xv □)
  (if (null? xs)
      (return (cons (cons nv □) (cons xv □)))
      (let  p1← □ in
        (if (< (car xs) nv)
          (mmp (cdr xs) □ (car xs) □ xv □))
            (if (> (car xs) xv)
              (mmp (cdr xs) □ nv □ (car xs) □)
              (mmp (cdr xs) □ nv □ xv □)))))
  (define (main)
   (return (mmp (cdr xs) □ (car xs) □ (car xs) □)))
  (main)
```

(b) Slice of program in (a) to compute only the min and max value

```
(define (mmp xs p nv np xv □)
  (if (null? xs)
      (return (cons (cons nv np) (cons xv □)))
      (let  p1← (+ p 1) in
        (if (< (car xs) nv)
          (mmp (cdr xs) p1 (car xs) p xv □))
          (if (> (car xs) xv)
              (mmp (cdr xs) p1 nv np (car xs) □)
              (mmp (cdr xs) p1 nv np xv □)))))
  (define (main)
   (return (mmp (cdr xs) 2 (car xs) 1 (car xs) □)))
  (main)
```

(c) Slice of program in (a) to compute the min value and its position.

Figure 5.4: A program in Scheme-like language and its slices. The parts that are sliced away are denoted by □.

mand at each expression π: e

2. Apply the Mohri-Nederhof procedure to construct the corresponding automaton $M_\pi^{\{\epsilon\}}$
3. A step called *canonicalization* which applies the simplification rules on $M_\pi^{\{\epsilon\}}$, but stops

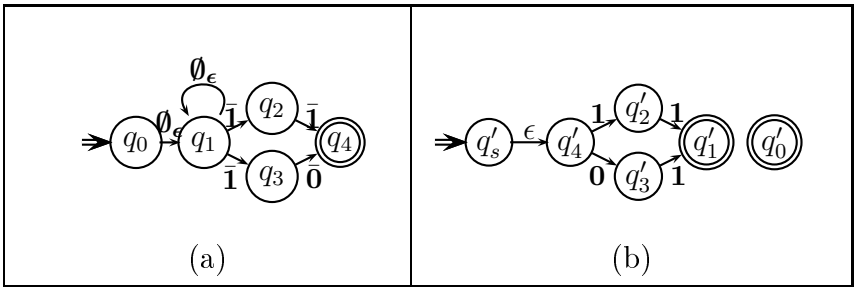


Figure 5.5: (a) The canonical automaton A_π and (b) the corresponding completing automaton $\overline{A}_\pi$

when the symbols $\bar{\mathbf{0}}$ and $\bar{\mathbf{1}}$ of every accepting string of the resulting automaton are only at the end

4. From the canonical automaton, constructing an automaton called the *completing automaton*, the output of the precomputation step

Since the first two steps have already been described in Chapter 3, we describe only the next two steps in detail.

5.2.2 Canonicalization

The *simplification* step defined in Section 3.2 reduces all strings which contains un-erased bar-edge symbols to null strings. The *canonicalization* step instead retains all strings that either have no $\bar{\mathbf{0}}$ and $\bar{\mathbf{1}}$ symbols or have $\bar{\mathbf{0}}$ and $\bar{\mathbf{1}}$ symbols only at the end. All bar-edge free strings correspond to expressions which will be present in the ϵ-slice and as a consequence in every non-trivial slice. Strings that have $\bar{\mathbf{0}}$ and $\bar{\mathbf{1}}$ symbols only at the end correspond to expressions which could potentially be part of a slice, given the right slicing criterion. We now give a set of rules, denoted by $\mathcal{C}$, that captures canonicalization.

$$\mathcal{C}(\{\epsilon\}) = \{\epsilon\} \qquad \mathcal{C}(\mathbf{0}\sigma) = \mathbf{0}\mathcal{C}(\sigma)$$
$$\mathcal{C}(\mathbf{1}\sigma) = \mathbf{1}\mathcal{C}(\sigma) \qquad \mathcal{C}(\mathbf{\emptyset_\epsilon}\sigma) = \mathbf{\emptyset_\epsilon}\mathcal{C}(\sigma)$$
$$\mathcal{C}(\bar{\mathbf{0}}\sigma) = \{\bar{\mathbf{0}} \mid \mathcal{C}(\sigma) \text{ is } \{\epsilon\}\} \cup \{\alpha \mid \mathbf{0}\alpha \in \mathcal{C}(\sigma)\}$$
$$\cup \{\bar{\mathbf{0}}\bar{\mathbf{1}}\alpha \mid \bar{\mathbf{1}}\alpha \in \mathcal{C}(\sigma)\} \cup \{\bar{\mathbf{0}}\bar{\mathbf{0}}\alpha \mid \bar{\mathbf{0}}\alpha \in \mathcal{C}(\sigma)\}$$
$$\mathcal{C}(\bar{\mathbf{1}}\sigma) = \{\bar{\mathbf{1}} \mid \mathcal{C}(\sigma) \text{ is } \{\epsilon\}\} \cup \{\alpha \mid \mathbf{1}\alpha \in \mathcal{C}(\sigma)\}$$
$$\cup \{\bar{\mathbf{1}}\bar{\mathbf{1}}\alpha \mid \bar{\mathbf{1}}\alpha \in \mathcal{C}(\sigma)\} \cup \{\bar{\mathbf{1}}\bar{\mathbf{0}}\alpha \mid \bar{\mathbf{0}}\alpha \in \mathcal{C}(\sigma)\}$$
$$\mathcal{C}(\sigma_1 \cup \sigma_2) = \mathcal{C}(\sigma_1) \cup \mathcal{C}(\sigma_2)$$

$\mathcal{C}$ differs from $\mathcal{S}$ in that it accumulates continuous run of $\bar{\mathbf{0}}$ and $\bar{\mathbf{1}}$ at the end of a string. Notice that $\mathcal{C}$, like $\mathcal{S}$, simplifies its input string from the right. Here is an example of $\mathcal{C}$ simplification:

$$\{\mathbf{0\emptyset_\epsilon 0\bar{1}\bar{1}1\bar{0}}\} \xrightarrow{\mathcal{C}} \mathbf{0}\mathcal{C}(\{\mathbf{\emptyset_\epsilon 0\bar{1}\bar{1}1\bar{0}}\}) \xrightarrow{\mathcal{C}} \mathbf{0\emptyset_\epsilon}\mathcal{C}(\{\mathbf{0\bar{1}\bar{1}1\bar{0}}\}) \xrightarrow{\mathcal{C}} \mathbf{0\emptyset_\epsilon 0}\mathcal{C}(\{\bar{\mathbf{1}}\bar{\mathbf{1}}\mathbf{1}\bar{\mathbf{0}}\})$$

$$\xrightarrow{\mathcal{C}} \mathbf{0\emptyset_\epsilon 0\bar{1}}\mathcal{C}(\{\bar{\mathbf{1}}\mathbf{1}\bar{\mathbf{0}}\}) \xrightarrow{\mathcal{C}} \mathbf{0\emptyset_\epsilon 0\bar{1}\bar{1}}\mathcal{C}(\{\mathbf{1}\bar{\mathbf{0}}\}) \xrightarrow{\mathcal{C}} \mathbf{0\emptyset_\epsilon 0\bar{1}\bar{1}1}\mathcal{C}(\{\bar{\mathbf{0}}\}) \xrightarrow{\mathcal{C}} \mathbf{0\emptyset_\epsilon 0\bar{1}\bar{1}1\bar{0}}\mathcal{C}(\{\epsilon\})$$

$$\xrightarrow{\mathcal{C}} \mathbf{0\emptyset_\epsilon 0\bar{1}\bar{1}1\bar{0}}\{\epsilon\} \xrightarrow{\mathcal{C}} \mathbf{0\emptyset_\epsilon 0\bar{1}\bar{1}1}\{\bar{\mathbf{0}}\} \xrightarrow{\mathcal{C}} \mathbf{0\emptyset_\epsilon 0\bar{1}\bar{1}}\{\mathbf{1}\bar{\mathbf{0}}\} \xrightarrow{\mathcal{C}} \mathbf{0\emptyset_\epsilon 0\bar{1}}\{\bar{\mathbf{0}}\}$$

$$\xrightarrow{\mathcal{C}} \mathbf{0\emptyset_\epsilon 0}\{\bar{\mathbf{1}}\bar{\mathbf{0}}\} \xrightarrow{\mathcal{C}} \mathbf{0\emptyset_\epsilon}\{\mathbf{0\bar{1}}\bar{\mathbf{0}}\} \xrightarrow{\mathcal{C}} \mathbf{0}\{\mathbf{\emptyset_\epsilon 0\bar{1}}\bar{\mathbf{0}}\} \xrightarrow{\mathcal{C}} \{\mathbf{0\emptyset_\epsilon 0\bar{1}}\bar{\mathbf{0}}\}$$

In contrast the simplification of the same string using $\mathcal{S}$ gives:

$$\{\mathbf{0\emptyset_\epsilon 0\bar{1}\bar{1}1\bar{0}}\} \xrightarrow{\mathcal{S}} \mathbf{0}\mathcal{S}(\{\mathbf{\emptyset_\epsilon 0\bar{1}\bar{1}1\bar{0}}\}) \xrightarrow{\mathcal{S}} \mathbf{0\emptyset_\epsilon}\mathcal{S}(\{\mathbf{0\bar{1}\bar{1}1\bar{0}}\}) \xrightarrow{\mathcal{S}} \mathbf{0\emptyset_\epsilon 0}\mathcal{S}(\{\bar{\mathbf{1}}\bar{\mathbf{1}}\mathbf{1}\bar{\mathbf{0}}\})$$

$$\xrightarrow{\mathcal{S}} \mathbf{0\emptyset_\epsilon 0\bar{1}}\mathcal{S}(\{\bar{\mathbf{1}}\mathbf{1}\bar{\mathbf{0}}\}) \xrightarrow{\mathcal{S}} \mathbf{0\emptyset_\epsilon 0\bar{1}\bar{1}}\mathcal{S}(\{\mathbf{1}\bar{\mathbf{0}}\}) \xrightarrow{\mathcal{S}} \mathbf{0\emptyset_\epsilon 0\bar{1}\bar{1}1}\mathcal{S}(\{\bar{\mathbf{0}}\}) \xrightarrow{\mathcal{S}} \mathbf{0\emptyset_\epsilon 0\bar{1}\bar{1}1\bar{0}}\mathcal{S}(\{\epsilon\})$$

$$\xrightarrow{\mathcal{S}} \mathbf{0\emptyset_\epsilon 0\bar{1}\bar{1}1\bar{0}}\emptyset \xrightarrow{\mathcal{S}} \mathbf{0\emptyset_\epsilon 0\bar{1}\bar{1}1}\emptyset \xrightarrow{\mathcal{S}} \mathbf{0\emptyset_\epsilon 0\bar{1}\bar{1}}\emptyset \xrightarrow{\mathcal{S}} \mathbf{0\emptyset_\epsilon 0\bar{1}}\emptyset \xrightarrow{\mathcal{S}} \mathbf{0\emptyset_\epsilon 0}\emptyset \xrightarrow{\mathcal{S}} \mathbf{0\emptyset_\epsilon}\emptyset \xrightarrow{\mathcal{S}} \mathbf{0}\emptyset \xrightarrow{\mathcal{S}} \emptyset$$

$\mathcal{C}$ satisfies two important properties:

1. The result of $\mathcal{C}$ always has the form $(\mathbf{0}+\mathbf{1}+\mathbf{\emptyset_\epsilon})^*(\bar{\mathbf{0}}+\bar{\mathbf{1}})^*$. Further, if $\sigma \subseteq (\mathbf{0}+\mathbf{1}+\mathbf{\emptyset_\epsilon})^*$, then $\mathcal{C}(\sigma) = \sigma$.
2. $\mathcal{S}$ subsumes $\mathcal{C}$, i.e., $\mathcal{S}(\mathcal{C}(\sigma_1)\mathcal{C}(\sigma_2)) = \mathcal{S}(\sigma_1\sigma_2)$.

Note that while we have defined canonicalization over a language, the actual canonicalization takes place over an automaton—specifically the automaton M_π obtained after Mohri-Nederhof transformation. The process of canonicalization over an automaton is a minor variation of the simplification process [44]. Specifically,

1. Adjacent $\bar{\mathbf{0}}\mathbf{0}$ and $\bar{\mathbf{1}}\mathbf{1}$ edges are replaced by an ϵ edge and the resulting automaton is made deterministic, until there are no more such edges
2. Edges with labels $\bar{\mathbf{0}}$ or $\bar{\mathbf{1}}$ are retained only if their targets have a path reaching some final node, and the labels on this path consist only of $\bar{\mathbf{0}}$ or $\bar{\mathbf{1}}$ symbols.

It is in the second step that the canonicalization differs from simplification. For the example program, the canonical automaton for π is shown in Figure 5.5a. Notice that all the strings in the language of $M_\pi^{\{\epsilon\}}$ will have the form $\emptyset_\epsilon{\emptyset_\epsilon}^*\bar{\mathbf{1}}\bar{\mathbf{1}}$ or $\emptyset_\epsilon{\emptyset_\epsilon}^*\bar{\mathbf{1}}\bar{\mathbf{0}}$. The $\bar{\mathbf{1}}$ and $\bar{\mathbf{0}}$ symbols are all towards the end of the string in both cases. Once all the automata have been converted into the canonical form, the next step is to convert each one of them into a completing automaton.

5.2.3 Completing automata generation

The completing automata generation step takes an automaton corresponding to expression e in canonical form, and computes an automaton which accepts all possible slicing criterion strings that keeps e in slice. For the motivating example 5.4, the automaton $M_\pi^{\{\epsilon\}}$ for the slicing criterion $\{\epsilon\}$ is shown in Figure 5.5a. In this automaton, each accepting string has $\bar{\mathbf{0}}$ and $\bar{\mathbf{1}}$ symbols only at the end. Thus the automaton is canonical, and we shall denote it as A_π. It is clear that if A_π is concatenated with a slicing criterion that starts with the symbol $\mathbf{01}$, the result, after simplification, will be non-empty, and the expression at π will be retained in the slice. We call such a string a *completing string* for A_π. Detecting the completing string became easy because the canonicalization step pushed all the $\bar{\mathbf{0}}$ and $\bar{\mathbf{1}}$ symbols towards the final state in the canonical automaton. Similarly, the string $\mathbf{11}$ is also a completing string for the same automaton.

Algorithm 9 describes the process for converting an automaton in canonical form to a completing automaton. The function **createCompletingAutomaton** takes A_π, the canonical Mohri-Nederhof automaton for the slicing criterion $\{\epsilon\}$, as input, and constructs the completing automaton, denoted as $\overline{A}_\pi$. Recollect that the strings recognized by A_π are of the form $(\mathbf{0}+\mathbf{1}+\emptyset_\epsilon)^*(\bar{\mathbf{0}}+\bar{\mathbf{1}})^*$. Call the set of states reachable from the start state using only edges with labels $\{\mathbf{0},\mathbf{1},\emptyset_\epsilon\}$ as the *frontier set*. The completing automaton is a copy of canonical automaton with edges labeled by $\bar{\mathbf{0}}$ and $\bar{\mathbf{1}}$ symbols reversed, and the symbols themselves replaced by $\mathbf{0}$ and $\mathbf{1}$ respectively. All edges with labels $\{\mathbf{0},\mathbf{1},\emptyset_\epsilon\}$ are dropped. Further, all states in the frontier set are marked as final states, and a new

Function createCompletingAutomaton(A)

Data: The Canonicalized Automaton $A = \langle Q, \{\mathbf{0}, \mathbf{1}, \bar{\mathbf{0}}, \bar{\mathbf{1}}, \boldsymbol{\emptyset_\epsilon}\}, \delta, q_0, F\rangle$

Result: $\overline{A}$, the completing automaton for A

$F' \leftarrow \{q_{\text{fr}} \mid q_{\text{fr}} \in Q, \mathsf{hasBarFreeTransition}(q_0, q_{\text{fr}}, \delta)\}$

/* Reverse the "bar" transitions: directions as well as labels */

foreach *(transition* $\delta(q, \bar{\mathbf{0}}) \rightarrow q'$*)***do**
add transition $\delta'(q', \mathbf{0}) \rightarrow q$

foreach *(transition* $\delta(q, \bar{\mathbf{1}}) \rightarrow q'$*)***do**
add transition $\delta'(q', \mathbf{1}) \rightarrow q$

$q'_s \leftarrow$ new state /* start state of $\overline{A}$ */

foreach *(state* $q \in F$*)***do**
add transition $\delta'(q'_s, \epsilon) \rightarrow q$

return $\langle Q \cup \{q'_s\}, \{\mathbf{0}, \mathbf{1}\}, \delta', q'_s, F'\rangle$

Function inSlice(e, σ)

Data: expression e, slicing criteria σ

Result: Decides whether e should be retained in slice

return $(\mathscr{L}(\overline{A}_e) \cap \ \sigma \neq \emptyset)$

Algorithm 9: Function to create the completing automaton and the slicing function.

start node is added with transitions to the states corresponding to the final states of canonical automaton. For the example program, the frontier set corresponding to $M_\pi^{\{\epsilon\}}$ is $\{q_0, q_1\}$ since these are the only states reachable from the start state using only edges with $\{\mathbf{0}, \mathbf{1}, \boldsymbol{\emptyset_\epsilon}\}$ labels. The completing automaton for π is the automaton in Figure 5.5b. After dropping edges with $\{\mathbf{0}, \mathbf{1}, \boldsymbol{\emptyset_\epsilon}\}$ symbols there is no path from the new state to the state corresponding to q_0. Since, state corresponding to q_1 is also final state, the completing automaton will have a non-null language.

Notice that for the canonical automata in Figure 5.5a, any string which has the prefix $(\mathbf{01} + \mathbf{11})$ is a valid completing string. Therefore, the automaton corresponding to the regular expression$\{(\mathbf{01} + \mathbf{11})(\mathbf{0} + \mathbf{1})^*\}$ recognizes *all* completing strings for A_π and nothing else. Thus for an arbitrary slicing criterion σ, it suffices to intersect σ with this automaton to decide whether the expression at π will be in the slice. In fact, it is enough for the completing automaton to recognize just the language $\{(\mathbf{01} + \mathbf{11})\}$ instead of $\{(\mathbf{01} + \mathbf{11})(\mathbf{0} + \mathbf{1})^*\}$. The reason is that any slicing criterion, say σ, is prefix closed,

and therefore $\sigma \cap \{(\mathbf{01} + \mathbf{11})\}$ is empty if and only if $\sigma \cap \{(\mathbf{01} + \mathbf{11})(\mathbf{0} + \mathbf{1})^*\}$ is empty. For our running example, the automaton in Figure 5.5b, gives the *completing automaton* that recognizes the language $(\mathbf{01} + \mathbf{11})$.

The incremental slicing algorithm uses the fact that a completing automaton accepts *all* slicing criterion strings that prevents the corresponding expression from being sliced. Whenever a slicing criterion is presented, we construct an automaton representing the criterion, which is then intersected with the completing automaton of every expression in the program. If the intersection turns out to be non-null, then the slicing criterion contains at least one string which would prevent the expression from being sliced. This fact is used to decide whether an expression can be sliced out or not. For the example program, we can see that the completing automaton corresponding to the program point π has a non-null intersection with the criterion $\{\epsilon, \mathbf{0}, \mathbf{00}, \mathbf{01}\}$ and hence it is retained in the slice, whereas it has a null intersection with $\{\epsilon, \mathbf{0}, \mathbf{1}, \mathbf{00}, \mathbf{10}\}$ allowing the corresponding expression to be sliced out. This matches our intuition that if neither the position of the minimum element nor the maximum element is required then the expression tracking the current position can be sliced out. We formally prove the correctness of incremental slicing in the next section.

5.3 Correctness of incremental slicing

We now show that the incremental algorithm to compute incremental slices is correct. Recall that we use the following notations:

1. G^{σ}_{π} is the grammar generated by dependence analysis (Figure 3.1) for an expression π: e in the program of interest, when the slicing criteria is σ
2. A_{π} is the automaton corresponding to $G^{\{\epsilon\}}_{\pi}$ after Mohri-Nederhof transformation and canonicalization
3. $\overline{A}_{\pi}$ is the completing automaton for e

We first show that the result of the dependence analysis for an arbitrary slicing criterion σ can be decomposed as the concatenation of the grammar obtained from the dependence analysis with the fixed slicing criterion $\{\epsilon\}$ and σ itself.

Lemma 5.4 *For all expressions e and slicing criteria σ, $\mathscr{L}(G^{\sigma}_{\pi}) = \mathscr{L}(G^{\{\epsilon\}}_{\pi})\,\sigma$.*

PROOF. The proof is by induction on the structure of e. Observe that all the rules of the dependence analysis (Figure 3.1) add symbols only as prefixes to the incoming demand. Hence, the slicing criteria will always appear as a suffix of any string that is produced by the grammar. Thus, any grammar $\mathscr{L}(G^{\sigma}_{\pi})$ can be decomposed as $\sigma'\,\sigma$ for some language σ'. Substituting $\{\epsilon\}$ for σ, we get $G^{\{\epsilon\}}_{\pi} = \sigma'$. Thus $\mathscr{L}(G^{\sigma}_{\pi}) = \mathscr{L}(G^{\{\epsilon\}}_{\pi})\,\sigma$. ■

Given a string s over $(\bar{\mathbf{0}}+\bar{\mathbf{1}})^*$, we use the notation $\overline{s}$ to stand for the reverse of s in which all occurrences of $\bar{\mathbf{0}}$ are replaced by $\mathbf{0}$ and $\bar{\mathbf{1}}$ replaced by $\mathbf{1}$. Clearly, $\mathcal{S}(\{s\overline{s}\}) = \{\epsilon\}$.

We next prove the completeness and minimality of $\overline{A}_{\pi}$.

Lemma 5.5 $\{s \mid \mathcal{S}(\mathscr{L}(M^{\{s\}}_{\pi})) \neq \emptyset\} = \mathscr{L}(\overline{A}_{\pi})\,(\mathbf{0}+\mathbf{1})^*$

PROOF. *We first prove LHS $\subseteq$ RHS. Let the string $s \in \mathcal{S}(\mathscr{L}(M^{\{s\}}_{\pi}))$. Then by Lemma 5.4, $s \in \mathcal{S}(\mathscr{L}(M^{\{\epsilon\}}_{\pi})\,\{s\})$. By Property 2, this also means that $s \in \mathcal{S}(\mathcal{C}(\mathscr{L}(M^{\{\epsilon\}}_{\pi}))\,\{s\})$. Since strings in $\mathcal{C}(\mathscr{L}(M^{\{\epsilon\}}_{\pi}))$ are of the form $(\mathbf{0}+\mathbf{1}+\boldsymbol{\emptyset}_{\boldsymbol{\epsilon}})^*(\bar{\mathbf{0}}+\bar{\mathbf{1}}))^*$ (Property 1), this means that there is a string $p_1\,p_2$ such that $p_1 \in (\mathbf{0}+\mathbf{1}+\boldsymbol{\emptyset}_{\boldsymbol{\epsilon}})^*$ and $p_2 \in (\bar{\mathbf{0}}+\bar{\mathbf{1}})^*$, and $\mathcal{S}(\{p_2\}\{s\}) \subseteq (\mathbf{0}+\mathbf{1})^*$. Thus s can be split into two strings s_1 and s_2, such that $\mathcal{S}(\{p_2\}\,\{s_1\}) = \{\epsilon\}$. Therefore $s_1 = \overline{p_2}$. From the construction of $\overline{A}_{\pi}$ we have $\overline{p_2} \in \mathscr{L}(\overline{A}_{\pi})$ and $s_2 \in (\mathbf{0}+\mathbf{1})^*$. Thus, $s \in \mathscr{L}(\overline{A}_{\pi})\,(\mathbf{0}+\mathbf{1})^*$.*

Conversely, for the proof of RHS $\subseteq$ LHS, we assume that a string $s \in \mathscr{L}(\overline{A}_{\pi})\,(\mathbf{0}+\mathbf{1})^*$. From the construction of $\overline{A}_{\pi}$ we have strings p_1, p_2, s' such that $p_1p_2 \in \mathcal{C}(\mathscr{L}(M^{\epsilon}_{\pi}))$, $p_1 \in (\mathbf{0}+\mathbf{1}+\boldsymbol{\emptyset}_{\boldsymbol{\epsilon}})^*$, $p_2 \in (\bar{\mathbf{0}}+\bar{\mathbf{1}})^*$, s is $\overline{p_2}s'$ and $s' \in (\mathbf{0}+\mathbf{1})^*$. Thus, $\mathcal{S}(\mathscr{L}(M^{\{s\}}_{\pi})) = \mathcal{S}(\mathscr{L}(M^{\{\epsilon\}}_{\pi}\,\{s\})) = \mathcal{S}(\mathcal{C}(\mathscr{L}(M^{\{\epsilon\}}_{\pi}))\{s\}\,) = \mathcal{S}(\{p_1\,p_2\,\overline{p_2}\,s'\}) = \{p_1s'\}$. Thus, $\mathcal{S}(\mathscr{L}(M^{\{s\}}_{\pi}))$ is non-empty and $s \in$ LHS. ■

We now prove our main result: Our slicing algorithm represented by **inSlice** (Algorithm 9) returns true if and only if $\mathcal{S}(\mathscr{L}(A^{\epsilon}_{\pi})\,\sigma)$ is non-empty.

Theorem 5.6 *The incremental slicing algorithm is sound i.e.*
$\mathcal{S}(\mathscr{L}(M^{\sigma}_{\pi})) \neq \emptyset \leftrightarrow$ **inSlice**(e, σ)

PROOF. We first prove the forward implication. Let $s \in \mathcal{S}(\mathscr{L}(M^{\sigma}_{\pi}))$. From Lemma 5.4, $s \in \mathcal{S}(\mathscr{L}(M^{\epsilon}_{\pi})\,\sigma)$. From Property 2, $s \in \mathcal{S}(\mathcal{C}(\mathscr{L}(M^{\epsilon}_{\pi}))\,\sigma)$. Thus, there are strings p_1, p_2 such that $p_1 \in \mathcal{C}(\mathscr{L}(M^{\epsilon}_{\pi}))$, $p_2 \in \sigma, s = \mathcal{S}(\{p_1p_2\})$. Further p_1 in turn can be decomposed as p_3p_4 such that $p_3 \in (\mathbf{0}+\mathbf{1}+\boldsymbol{\emptyset}_{\boldsymbol{\epsilon}})^*$ and $p_4 \in (\bar{\mathbf{0}}+\bar{\mathbf{1}})^*$. We also have $\mathcal{S}(\{p_4p_2\}) \subseteq (\mathbf{0}+\mathbf{1})^*$. Thus $\overline{p_4}$ is a prefix of p_2.

From the construction of $\overline{A}_\pi$, we know $\overline{p_4} \in \mathscr{L}(\overline{A}_\pi)$. Further, $\overline{p_4}$ is a prefix of p_2 and $p_2 \in \sigma$, from the prefix closed property of σ we have $\overline{p_4} \in \sigma$. This implies $\overline{A}_\pi \cap \sigma \neq \emptyset$ and thus **inSlice**(e, σ) returns true.

Conversely, if **inSlice**(e, σ) is true, then $\exists s : s \in \mathscr{L}(\overline{A}_\pi) \cap \sigma$. In particular, $s \in \mathscr{L}(\overline{A}_\pi)$. Thus, from Lemma 5.5 we have $\mathcal{S}(\mathscr{L}(M_\pi^{\{s\}})) \neq \emptyset$. Further, since $s \in \sigma$ we have $\mathcal{S}(\mathscr{L}(M_\pi^\sigma)) \neq \emptyset$. ■

Table 5.1: Statistics for incremental and non-incremental slicing

Program	Precomputation time (ms)	#exprs in program	Slicing with $\{\epsilon\}$			Slicing with $\{\epsilon, 0\}$			Slicing with $\{\epsilon, 1\}$		
			Non-inc time (ms)	Inc time (ms)	#expr in slice	Non-inc time (ms)	Inc time (ms)	#expr in slice	Non-inc time (ms)	Inc time (ms)	#expr in slice
First-order Programs											
treejoin	6900.0	581	6163.2	2.4	536	5577.2	2.8	538	5861.4	4.6	538
deriv	399.6	389	268.0	1.6	241	311.2	1.6	249	333.2	2.3	266
paraffins	3252.8	1152	2287.3	5.2	1067	2529.2	5.1	1067	2658.7	5.1	1067
nqueens	395.4	350	309.9	1.5	350	324.6	1.5	350	328.1	1.6	350
minmaxpos	27.9	182	18.1	0.9	147	19.5	0.8	149	20.5	0.9	149
nperm	943.1	590	627.4	2.1	206	698.4	11.2	381	664.0	11.8	242
linecharcount	11.7	91	7.0	0.5	69	7.5	0.5	78	7.4	0.5	82
studentinfo	1120.6	305	858.2	1.2	96	854.6	1.3	101	1043.3	7.5	98
knightstour	2926.5	630	2188.1	2.8	436	2580.6	12.2	436	2492.8	7.4	436
takl	71.6	151	46.1	0.7	99	49.5	0.8	105	48.5	0.7	99
lambda	4012.9	721	3089.0	2.7	26	3377.4	13.2	705	2719.8	5.3	33

5.4 Experiments and results

In this section, we present the results obtained from our implementation of the slicing algorithm described. Since slicing time and accuracy (number of expressions sliced) are not reported for other slicing methods, we implemented both an incremental slicer and a non-incremental slicer and compared the two. The non-incremental slicing part was implemented as part of a BTP [86]. The non-incremental version does not construct completing automatons and hence needs to simplify automatons at each program point for every slicing criteria. Our experiments show that the incremental slicing algorithm is efficient when the overhead of creating the completing automata is amortized over the computation of a number of slices with different criteria.

For each slicing criteria, we compare the times for incremental step and computed non-incremental slicing. The results in Table 5.1 show that for all benchmarks the time required to compute the completing automatons is less than twice the time for slicing non-incrementally. The results confirm our hypothesis that incremental slicing is orders of magnitude faster than non-incremental slicing.

5.5 Static slicing of higher-order programs

We now describe a method using which our dependence analysis can be extended to handle higher-order programs. We achieve this by first converting the input higher-order program to its equivalent first-order program by a process called firstification [60]. During the firstification process we maintain a mapping between the original program and the firstified program. We perform dependence analysis on the firstified program and obtain the results. Using the mapping generated during the firstification process, we transfer the demands generated on the firstified program to the original program. We describe this method with an example for extending our static slicing algorithm to handle higher-order programs.

We now describe, using an example, how our method can be extended to handle slicing of higher order programs. While this is work in progress, our description will make it clear that the extension is implementable. Our example for this section will be the program in Figure 5.6a. It contains a higher order function **hof** which takes a function `f` and a second argument `lst` and applies `f` to `lst`. The function **main** creates a list `lst1`

```
(define (hof f lst) (return π:(f lst)))
(define (foldr f id lst)
    (if(null? lst)) (return id)
      (return (f (car lst) (foldr f id (cdr lst)))))
(define (fun x y) (return (+ y 1)))
(define (main)
    (let lst1 ← (cons a (cons b nil)) in
      (let g ← (foldr fun 0) in
        (return (cons (hof car lst1) (hof g lst1)))))
(main)
```

(a) A program with higher order functions.

```
(define (hof_g l) (return π_f:(foldr_fun 0 l)))
(define (hof_car lst) (return π_c:(car lst)))
(define (foldr_fun id lst)
    (if(null? lst)) (return id)
      (return (fun (car lst) (foldr_fun id (cdr lst)))))
(define (fun x y) (return (+ y 1)))
(define (main)
    (let lst1 ← (cons a (cons b nil)) in
      (let g ← (foldr_fun 0) in
        (return (cons (hof_car lst1) (hof_g lst1))))))
(main)
```

(b) Program after specialization.

```
(define (hof f lst) (return π:(f lst)))
(define (main)
    (let lst1 ← (cons a □) in
      (let g ← □) in
        (return (cons (hof car lst1) □))))
(main)
```

(c) Slice of program in (a) with slicing criterion $\{\epsilon, \mathbf{0}\}$.

Figure 5.6: An example higher order program

and a function value through a partial application and binds it to **g**. It makes two calls to **hof**. The first call to **hof** uses **car** and `lst1` as arguments and the second call uses the built-in function **even** and (**g** `lst1`) as arguments. Finally, **main** returns a **cons** cell created from the result of these calls.

We first discuss how demands are propagated from a call to its arguments in the case of a higher-order function. Observe that in Figure 5.6a the demands on the second argument to **hof** depends not only on the demand on **hof** but also on the function being passed as an argument to **hof**. To handle this, we specialize **hof** using the function argument in a manner similar to [60]. We create two specialized versions **hof_car** and **hof_g** corresponding to the two calls. The specialized program for our example is shown in Figure 5.6b.

We can now find out the demands on each expression of the specialized program using our dependence analysis. Moreover, the separation of the two calls to **hof** through specialization adds precision to the analysis. In the specialized program, the demands on the arguments `lst1` and (**g** `lst1`) now come separately from **hof_car** and **hof_g** and are not merged.

The body of **hof**, on the other hand, gets its demand from both the specialized calls. Hence, we maintain a mapping from each higher order function to all its first order variants. Once the demands for all the first order functions are computed, this mapping is used to compute the demands for the body of the higher order function. In the example, we maintain the mapping $\pi \mapsto \{\pi_c, \pi_f\}$. The demand on π is given by the union of the demands on π_c and π_f.

Even after specialization, partial applications, such as (**foldr_fun** 0), may remain in the residual program. Whenever a functional value is created via partial application it needs to maintain information about the first order *base* function using which the functional value gets created. As an example, to compute the demand on `lst1` in the specialized program, it is necessary to know that **foldr_fun** is the base for **g** and `lst1` is the second argument to **foldr_fun**. The demand on the effective first order call (**foldr_fun** 0 `lst1`) is obtained from the demand on (**g** `lst1`).

The actual process of slicing remains same. At each program point in the higher order function we store the completing automaton and whenever a slicing criterion is applied we just check for the intersection. For our example, given a slicing criterion $\{\epsilon, \mathbf{0}\}$, the sliced

program is shown in Figure 5.6c. The specialization enables computation of context-independent-summaries, even in the presence of higher order functions. As a result, the **cdr** part of `lst1` gets sliced away.

5.5.1 Limitations

Note that our simple firstifier requires us to statically find all bindings of a functional parameter. This is not possible if we allow functions to be returned as results or store functions in data-structures. As an example we can consider a function f, that, depending on a calculated value n, returns a function g iterated n times (i.e. $g \circ g \circ \overset{n \ times}{\ldots} \circ g$). A higher-order function receiving this value as a parameter cannot be specialized using the techniques described, for example, in [60]. A similar situation can show if we allow functions in lists.

5.6 Related work

Most of the efforts in slicing have been for imperative programs. The surveys [16, 92, 97] give good overviews of the variants of the slicing problem and their solution techniques. In the context of imperative programs, a slicing criterion is a pair consisting of a program point, and a set of variables. The slicing problem is to determine those parts of the program that decide the values of the variables at the program point [103]. A natural solution to the slicing problem is through the use of data and control dependences between statements. Thus the program to be sliced is transformed into a graph called the program dependence graph (PDG) [39, 72], in which nodes represent individual statements and edges represent dependences between them. The slice consists of the nodes in the PDG that are reachable through a backward traversal starting from the node representing the slicing criterion. Horwitz, Reps and Binkley [39] extend PDGs to handle interprocedural slicing. They show that a naive extension could lead to imprecision in the computed slice due to the incorrect tracking of the calling context. Their solution is to construct a context-independent summary of each function through a linkage grammar, and then use this summary to process function calls. The resulting graph is called a system dependence graph (SDG). Our method generalizes SDGs to additionally keep track of the construction of algebraic data types (**cons**), selection of components of data types (**car** and **cdr**) and

their interactions (the **cons-car** and **cons-cdr** cancellations), which may span across function boundaries.

Silva, Tamarit and Tomás [93] adapt SDGs for functional languages, in particular Erlang. The adaptation is straightforward except that they handle dependences that arise out of pattern matching. Because of the use of SDGs, they can manage calling contexts precisely. However, as pointed out by the authors themselves, when given the Erlang program: {**main**() -> x = {1,2}, {y,z} = x, y}, their method produces the imprecise slice {**main**() -> x = {1,2}, {y,□} = x, y} when sliced on the variable y. Notice that the slice retains the constant 2, and this is because of inadequate handling of the interaction between **cons** and **cdr**. For the equivalent program (**let** x← (**cons** 1 2) **in** (**let** y ← (**car** x) **in** y)) with the slicing criterion ϵ, our method would correctly compute the demand on the constant 2 as $\bar{\mathbf{1}}(\epsilon \cup \mathbf{0})$. This simplifies to the demand $\emptyset$, and 2 would thus not be in the slice. Another issue is that while the paper mentions the need to handle higher order functions, it does not provide details regarding how this is actually done. This would have been interesting, given that the language considered allows lambda expressions.

The slicing technique that is closest to ours is due to Reps and Turnidge [79]. They use projection functions, represented as tree grammars, as slicing criteria. Given a program P and a projection function ψ, their goal is to produce a program which behaves like $\psi \circ P$. Their analysis consists of propagating the projection function backwards to all subexpressions of the program. After propagation, any expression with the projection function $\perp$ (corresponding to our $\emptyset$ demand), is sliced out of the program. Liu and Stoller [57] use a similar method for dead code analysis and elimination. As shown earlier, our slicing algorithm subsumes dead code elimination.

These techniques differ from ours in two respects. These methods, unlike ours, do not derive context-independent summaries of functions. This results in a loss of information due to merging of contexts and affects the precision of the slice. As mentioned earlier, the computation of function summaries using symbolic demands enables the incremental version of our slicing method. Consider, as an example, the program fragment $\pi\colon(\mathbf{cons}\ \pi_1\colon x\ \pi_2\colon y)$ representing the body of a function. Dependence analysis with the symbolic demand σ gives the demand environment $\{\pi \mapsto \sigma, \pi_1 \mapsto \bar{\mathbf{0}}\sigma, \pi_2 \mapsto \bar{\mathbf{1}}\sigma\}$. Notice that the demands π_1 and π_2 are in terms of the symbols $\bar{\mathbf{0}}$ and $\bar{\mathbf{1}}$. This is a result of

our decision to work with symbolic demands. If we now slice with the default criterion ϵ and then canonicalize (instead of simplify), we are left with the demand environment $\{\pi \mapsto \epsilon, \pi_1 \mapsto \bar{\mathbf{0}}, \pi_2 \mapsto \bar{\mathbf{1}}\}$. There is enough information in the demand environment to deduce that π_1 (π_2) will be in the slice only if the slicing criterion includes $\mathbf{0}(\mathbf{1})$. Since the methods in [79] and [57] deal with demands in their concrete forms, it is difficult to see the incremental version being replayed with their methods.

There are other less related approaches to slicing. A graph based approach has also been used by Rodrigues and Barbosa [80] for component identification in Haskell programs. Given the intended use, the nodes of the graph represents coarser structures such as modules, functions and data type definitions, and the edges represents relations such as containment (e.g. a module containing a function definition). On a completely different note, Rodrigues and Barbosa [81] use program calculation in the Bird-Meerteens formalism for obtaining a slice. Given a program P and a projection function ψ, they calculate a program which is equivalent to $\psi \circ P$. However the method is not automated. Finally, dynamic slicing techniques have been explored for functional programs by Perera et al. [73], Ochoa et al. [67] and Biswas [17].

Chapter 6

Conclusions and future work

In this thesis we have defined a dependence analysis that is context-sensitive and takes into account structure transmitted data dependences, i.e. dependences arising from selector-constructor cancellation rules. We provide a formal definition of dependence analysis using an operational semantics called Demand Guided Semantics (DGS). In addition to the normal execution state transitions, DGS also specifies how demand on (**main**) propagates to constituent sub-expressions of the program. While this has been proved by Reps [78] in a different setting, we independently prove that computing fully precise dependence information as formulated in this thesis is undecidable. Reps shows that context-sensitivity and structure transmitted dependence can be modelled as context-free grammars individually. Modelling both at the same time is equivalent to finding out whether the intersection of two CFGs is empty, which is known to be undecidable.

Based on our formulation of dependence analysis, we come up with an algorithm that computes an over-approximation of dependences. We believe our analysis to be more precise than [79], which is not context-sensitive, and [93] which is context-sensitive but does not precisely model constructor-selector cancellation. The analysis defined in this thesis is more precise because instead of sacrificing either context-sensitivity or precise modelling of structure transmitted data dependence, we over approximate the context-sensitivity by a regular grammar instead of a CFG. Since the emptiness question for an intersection of regular grammar and a CFG is known to be decidable [64] our analysis captures context-sensitivity precisely when the grammar is already regular and only over approximates if it is a CFG. We show that the approximate dependence analysis is sound with respect to DGS.

We show the usefulness of dependence analysis in two applications, namely, liveness-based garbage collection in lazy languages and static program slicing. While liveness-based garbage collection has been shown to be effective in eager languages [12], this is the first attempt to handle lazy languages. Since it is difficult to determine exactly when an expression will be evaluated in a lazy language, deciding when to declare a variable dead becomes difficult. This is further aggravated by the fact that lazy constructors carry references to free variables in closures (which we call closure variables) outside the scope in which the variables were declared. This forces the following design decisions with respect to liveness analysis, 1) We introduce the notion of closure variables and treat them as first-class citizens from the perspective of garbage collection by extending the root set (variables on the program stack) to include closure variables and 2) We carry liveness of closure variables as part of the closure. The design of the garbage collector also becomes more complex as the collector has to deal with unevaluated expressions (closures) along with data values. We have proved the correctness of the liveness-based garbage collection scheme for a lazy language.

We model slicing as a dependence analysis problem by viewing the slicing criterion (a set of strings over $(\mathbf{0} + \mathbf{1})^*$) as a demand on the main expression (**main**) and use it to compute the demand on each expression in the program. Any expression which gets a $\emptyset$ demand can then be sliced away. Since applications such as program specialization and parallelization require the same program to be sliced with more than one slicing criteria. We propose an efficient way of slicing called *incremental slicing*. The incremental algorithm avoids repeated computations of dependences by a one time precomputation that computes for every expression *all* slicing criteria that keep the expression in the slice. The interesting fact is that such a set can be efficiently computed and is related to our decision of computing function summaries in terms of symbolic demands. Since the same program is sliced multiple times, the cost of the precomputation step gets amortized whenever the program is sliced with a different criterion.

6.1 Future work

The work presented in this thesis can be extended in multiple ways. In the formulation of dependence analysis problem proposed in Chapter 2, the designated expression on which

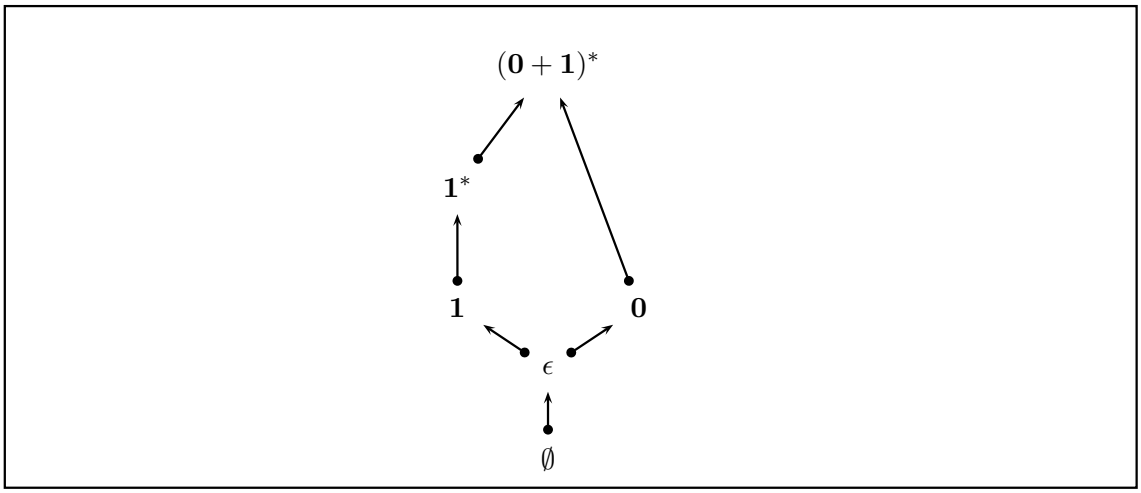


Figure 6.1: Lattice of demands

the user places the external demand is always (**main**). However, one may relax this restriction and an arbitrary expression $\hat{e}$ can be chosen as the designated expression on which the user places the demand $\hat{\sigma}$. This is very similar to the way the problem is posed for imperative languages. To handle this situation, we introduce the following rules that determine the demand on the body of each function in the program.

1. The demand on the main expression (**main**) is $\emptyset$.
2. The demand on a function body e_f is the union of demand over all calls to f. As a consequence, the demand on $e_{\mathbf{main}}$ is also $\emptyset$.
3. The demand propagation on the designated expression is carried out using $\hat{\mathcal{D}}$ instead of $\mathcal{D}$. These functions "inject" a demand of $\hat{\sigma}$ over the demand that reaches the designated expression. Formally:

$$\hat{\mathcal{D}}(\hat{s}, \sigma, \mathbb{DS}) = \mathcal{D}(\hat{s}, \sigma \cup \hat{\sigma}, \mathbb{DS})$$

Notice that our earlier formulation is a special case of the rules above when the designated expression is (**main**).

6.1.1 Liveness-based garbage collection

Although liveness-based garbage collection is efficient in collecting more garbage per collection than reachability-based collectors, the total time spent in doing garbage collection for liveness-based collectors does not compare favourably with reachability-based collectors. As mentioned in Section 4.4.3 and [12], this is mainly due to liveness-based collectors

```
(define (append x y)
  (if (eq? x nil)
    (return y)
    (let a ← (cdr x) in
      (let b ← (append a y) in
    (let u ← (car x) in
          (let w ← (cons u b) in
            (return w))))))
```

(append x y)	x	y
$\emptyset$	$\emptyset$	$\emptyset$
ϵ	ϵ	ϵ
$\mathbf{0}$	$\mathbf{0}$	$\mathbf{0}$
$\mathbf{1}$	$\mathbf{1}^*$	$\mathbf{1}$
$\mathbf{1}^*$	$\mathbf{1}^*$	$\mathbf{1}^*$
$(\mathbf{0}+\mathbf{1})^*$	$(\mathbf{0}+\mathbf{1})^*$	$(\mathbf{0}+\mathbf{1})^*$

Figure 6.2: Function **append** and its function-summary table.

traversing the same memory locations multiple times where a reachability-based collector would do a single traversal. For liveness-based garbage collection scheme to become mainstream, the efficiency of the analysis and the garbage collector itself have to be improved. Another drawback of liveness-based collectors is the extra memory required for storing liveness automata. We discuss some ideas to mitigate these drawbacks.

Improve performance of liveness-based garbage collector

Our experiments with liveness-based garbage collection suggest that most programs do not require the kind of precise liveness generated by our analysis. Sufficient gains can be made over a reachability-based collector even when we restrict our liveness values to a small set of liveness values. We borrow ideas from strictness analysis and restrict the possible liveness values to a finite set of patterns. By sacrificing some precision, both the analysis and the process of garbage collection itself can be made faster. Figure 6.1 shows the lattice of allowed liveness values in our analysis. The input and output values can only be one among the values represented in the lattice. While we describe our method for a lazy first-order language, the same is applicable to eager languages.

Consider the example shown in Figure 6.2, we show how to compute the demand transformer for the function **append** with the restricted set of demands. Since we are dealing with a finite set of demands only, computing context-independent summaries becomes simpler. We take each demand in the finite set and use it as a concrete demand on the function body and do a fixed-point computation to obtain the demand transformer. The fixed-point computation can be done as shown by the example in Figure 6.3. Assume that we need to compute the demand transformer $\mathbb{DS}^{\mathbf{1}}_{\mathbf{append}}$, we first assume that the

Iteration #	Assumed value for $\mathbb{DS}^{\mathbf{1}}_{\mathbf{append}}$	$\emptyset_{\epsilon}\sigma \cup \mathbf{1}\mathbb{DS}^{\mathbf{1}}_{\mathbf{append}}(\bar{\mathbf{1}}\sigma) \cup \mathbf{0}\bar{\mathbf{0}}\sigma$
0	$\emptyset \mapsto \emptyset$ $\epsilon \mapsto \emptyset$ $\mathbf{0} \mapsto \emptyset$ $\mathbf{1} \mapsto \emptyset$ $\mathbf{1}^* \mapsto \emptyset$ $(\mathbf{0}+\mathbf{1})^* \mapsto \emptyset$	$\emptyset \mapsto \emptyset$ $\epsilon \mapsto \epsilon$ $\mathbf{0} \mapsto (\mathbf{0}+\mathbf{1})^*$ $\mathbf{1} \mapsto \epsilon$ $\mathbf{1}^* \mapsto \epsilon$ $(\mathbf{0}+\mathbf{1})^* \mapsto (\mathbf{0}+\mathbf{1})^*$
1	$\emptyset \mapsto \emptyset$ $\epsilon \mapsto \epsilon$ $\mathbf{0} \mapsto (\mathbf{0}+\mathbf{1})^*$ $\mathbf{1} \mapsto \epsilon$ $\mathbf{1}^* \mapsto \epsilon$ $(\mathbf{0}+\mathbf{1})^* \mapsto (\mathbf{0}+\mathbf{1})^*$	$\emptyset \mapsto \emptyset$ $\epsilon \mapsto \epsilon$ $\mathbf{0} \mapsto (\mathbf{0}+\mathbf{1})^*$ $\mathbf{1} \mapsto \mathbf{1}^*$ $\mathbf{1}^* \mapsto \mathbf{1}^*$ $(\mathbf{0}+\mathbf{1})^* \mapsto (\mathbf{0}+\mathbf{1})^*$
2	$\emptyset \mapsto \emptyset$ $\epsilon \mapsto \epsilon$ $\mathbf{0} \mapsto (\mathbf{0}+\mathbf{1})^*$ $\mathbf{1} \mapsto \mathbf{1}^*$ $\mathbf{1}^* \mapsto \mathbf{1}^*$ $(\mathbf{0}+\mathbf{1})^* \mapsto (\mathbf{0}+\mathbf{1})^*$	$\emptyset \mapsto \emptyset$ $\epsilon \mapsto \epsilon$ $\mathbf{0} \mapsto (\mathbf{0}+\mathbf{1})^*$ $\mathbf{1} \mapsto \mathbf{1}^*$ $\mathbf{1}^* \mapsto \mathbf{1}^*$ $(\mathbf{0}+\mathbf{1})^* \mapsto (\mathbf{0}+\mathbf{1})^*$

Figure 6.3: Table showing the fixed-point computation for computing the demand transformer for `x`, the first argument of **append**. In each iteration, the assumed value is substituted in the equation in the third column to get the actual demand on `x`.

demand transformer takes any demand and returns $\emptyset$. Using this we compute the demand on `x` which is given by the equation,

$$\mathbb{DS}^{\mathbf{1}}_{\mathbf{append}}(\sigma) = \emptyset_{\epsilon}\sigma \cup \mathbf{1}\mathbb{DS}^{\mathbf{1}}_{\mathbf{append}}(\bar{\mathbf{1}}\sigma) \cup \mathbf{0}\bar{\mathbf{0}}\sigma$$

Using the assumed function for $\mathbb{DS}^{\mathbf{1}}_{\mathbf{append}}$ in the RHS, we get our next approximation. For example, for the demand $\mathbf{1}^*$, we get $\mathbb{DS}^{\mathbf{1}}_{\mathbf{append}}(\mathbf{1}^*) = \{\epsilon\}$. We use the new set of mappings to get the next approximation where again some values get updated. Considering the same input demand $\mathbf{1}^*$ as before, the new mapping is $\mathbb{DS}^{\mathbf{1}}_{\mathbf{append}}(\mathbf{1}^*) = \{\epsilon\} \cup \{\mathbf{1}\}$. Since, this value is not part of the finite liveness values being considered we replace it with a

value in the lattice which contains both $\{\epsilon\}$ and $\{\mathbf{1}\}$ i.e. $\mathbf{1}^*$. Repeating the process once more, we find that there are no more changes in the mapping. We stop the iteration as we have reached a fixed-point. The mapping obtained at the end of the iteration is the required demand transformer $\mathbb{D}\$^{\mathbf{1}}_{\mathbf{append}}$.

Working with a finite set of liveness values avoids the simplification/erasure process and thus improves the efficiency of computing liveness automata. Whenever we encounter a function call with demand σ, we look-up the table corresponding to the function and use the argument demands corresponding to σ. Another factor which makes liveness-based collection slow is the need for consulting liveness automata at each step during garbage collection. By using a finite set of liveness values we can hardcode the processing for each liveness value in the garbage collector. The garbage collector can then just check the liveness value associated with the root variable and call the specialized code for handling that liveness value. This not only saves time but also memory by avoiding the storing of liveness automata completely. Liveness values now can be embedded inside the heap cell itself as an enum.

Finally, since the fixed set of values form a lattice, whenever a heap cell has been visited with a value which contains the incoming value we can avoid traversing the heap again. For example, if it is known that a reference has been traversed using the liveness value $(\mathbf{0}+\mathbf{1})^*$, then if the same root cell is being traversed with the value $\mathbf{1}^*$ the repeated traversal can be avoided.

In case of a lazy language we can further take advantage by reducing the extra memory required for storing liveness automata references for closure arguments. Each closure need only store an enumeration corresponding to the liveness value of each of its argument.

Extending demand propagation to higher order programs

Our current way of handling higher-order programs requires the program to be converted to a first-order program. This can be avoided if we have an analysis which can handle higher-order programs. By restricting the demands to a finite set, we can extend our analysis to handle higher-order programs.

In the case of a higher-order function, the demand on the non-functional argument can depend on the functional argument. The first step in determining the demand on

```
(define (foldr f id lst)
  (if(null? lst)) (return id)
    (return (f (car lst)
               (foldr f
                      id (cdr lst)))))
```

(a)

$\emptyset \to \emptyset$ $\quad \emptyset \to \emptyset$ $\quad \emptyset \to \epsilon$
$\epsilon \to \emptyset$ $\quad \epsilon \to \epsilon$ $\quad \epsilon \to \epsilon$

(b)

Demand on call to **foldr**	Demand on `id`	Demand on `lst`
$\emptyset$	$\emptyset$	$\emptyset$
$\{\epsilon\}$	$\{\epsilon\}$	$\mathbf{1}^*$

(c)

Figure 6.4: (a) Definition of **foldr** (b) Potential demand transformers for the first argument of **f** when **f** is restricted to functions that take integer arguments and return an integer. (c) Function summary table corresponding to **foldr** when **foldr** is used to compute length of `lst`.

the non-functional argument is to find out all potential demand transformers for the higher-order argument and use them to create a table.

Consider the higher-order function **foldr** in Figure 6.4(a). Restricting the argument **f** to functions which take integer values and return an integer we can generate the summaries for functions that could be passed to **foldr**. Figure 6.4(b) lists the possible function summaries for the first argument of **f**. Since **f** takes two arguments, and the second argument also has similar potential transformers, considering all possible combinations we will have 9 potential function-summaries for **f**. Once all the possible function summaries have been generated, we generate the function summary for **foldr** considering each possible function summary for **f**. During analysis when a higher-order function call is encountered, we can use the summary for the actual function being passed and compute the demands on the arguments of **foldr**. As an example, consider the implementation of **length** function using **foldr**, given as (**foldr** (`a b` (+ 1 `b`) 0 `xs`). The anonymous function passed has demand-transformer which maps both demands $\emptyset$ and $\{\epsilon\}$ to $\emptyset$ for its first argument and the transformer for the second argument maps $\emptyset$ to $\emptyset$ and $\{\epsilon\}$ to $\{\epsilon\}$. The

function summary table for **foldr** for this function is shown in Figure 6.4(c). Since, the passed function never uses its first argument, the demand on the elements of `xs` is always $\emptyset$. Thus, the demand transformer corresponding to `xs` transforms an $\{\epsilon\}$ demand on a call to **foldr** into a $\mathbf{1}^*$ demand on `xs`.

Further, demand on the functional argument has to be added to the demand-summary of the actual function being passed. Demands on the expressions inside the body of **foldr** are computed as usual by considering the union of demands at all call points. The major challenge in using this method is to handle the huge number of potential demand transformers that needs to be considered for each higher-order function.

Hybrid GC - (reachability and liveness)

Another way to improve the effectiveness of garbage collection is to have a hybrid garbage collector which can invoke either reachability or liveness-based collector. The intuition behind this idea is that running a slow liveness-based collection is justified only if there are sufficiently large number of reachable but dead cells. Therefore, we invoke a liveness-based collector once after every k invocations of a fast reachability-based collector. Over several runs of a reachability-based collector sufficient memory which is reachable but not live gets accumulated and hence running a liveness-based collector will give sufficient advantage. The cost of invoking a liveness-based collector is thus amortized over several calls of a reachability-based collector. However, care has to be taken to ensure that after a liveness-based collection, any references which point to the dead part of the heap are correctly nullified.

k-Liveness GC

This is an extension of the idea due to Agesen [6] which just tests for the liveness of root variables in Java. Instead of checking liveness of just the root variable, we use liveness upto k levels. Beyond k levels everything that is reachable is copied. We can avoid repeated traversals by maintaining an extra bit in each heap cell which can be set if it was copied using reachability. During a traversal, if this bit is set the collector need not traverse the substructure. The only drawback in this approach is that even for common functions like **length** which only traverses the spine of its argument, a k-liveness collector could end up copying extra cells.

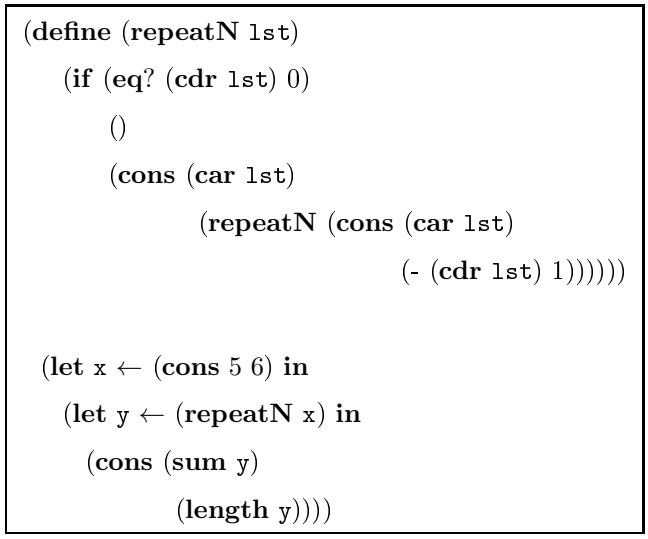


Figure 6.5: Motivating example for forward demand propagation. If we take a forward slice with respect to **car** part of `x` it can be seen that the expression (**length** `y`) can be sliced.

6.1.2 Forward slicing using demand propagation

The dependence analysis defined in this thesis is a backward analysis i.e. it takes a demand on the result of an expression and computes the demand on the arguments of the expression. This allowed us to use the result of the analysis to solve problems such as computing liveness and backward slicing. There are applications which can benefit from an analysis which takes demands on the arguments of an expression and computes demands on the output of the expression i.e. a forward analysis. A forward version of our dependence analysis would be a good extension. The example in Figure 6.5 shows how forward analysis is useful. In the example, let us assume that we are interested in knowing what parts of the final output might be affected when we modify the value of the **car** part of `x`(5). We can take a forward slice of the program with respect to the **car** part of `x`. The forward slice thus obtained will not contain the **cdr** part of the final output. This is intuitive, as the length of the list produced by the function **repeatN** depends only on the **cdr** part of `x`.

Bibliography

[1] Program slice browser. In *Proceedings of the 9th International Workshop on Program Comprehension* (Washington, DC, USA, 2001), IWPC '01, IEEE Computer Society, pp. 50–.

[2] An artificial garbage collection benchmark. `http://www.hboehm.info/gc/gc_bench.html`, Nov 2015. (Last Accessed).

[3] Huffman encoding trees. `https://mitpress.mit.edu/sicp/full-text/sicp/book/node41.html`, Nov 2015. (Last Accessed).

[4] PLT Scheme Benchmark Suite. `http://svn.plt-scheme.org/plt/trunk/collects/tests/`, Nov 2015. (Last accessed).

[5] Agesen, O., Detlefs, D., and Moss, J. E. Garbage collection and local variable type-precision and liveness in java virtual machines. *SIGPLAN Not. 33*, 5 (May 1998), 269–279.

[6] Agesen, O., Detlefs, D., and Moss, J. E. Garbage collection and local variable type-precision and liveness in Java virtual machines. In *PLDI* (1998).

[7] Allen, F. E. Control flow analysis. In *Proceedings of a Symposium on Compiler Optimization* (New York, NY, USA, 1970), ACM, pp. 1–19.

[8] Allen, F. E. A basis for program optimization. In *IFIP Congress (1)* (1971), pp. 385–390.

[9] Andersen, L. O. Program analysis and specialization for the c programming language. Tech. rep., 1994.

[10] ANIELLO, C., DE, L. A., AND MALCOLM, M. A specification driven slicing process for identifying reusable functions. *Journal of Software Maintenance: Research and Practice 8*, 3, 145–178.

[11] APPEL, A. W. *Compiling with Continuations*. Cambridge University Press, New York, NY, USA, 2007.

[12] ASATI, R., SANYAL, A., KARKARE, A., AND MYCROFT, A. Liveness-based garbage collection. In *Compiler Construction - 23rd International Conference, CC 2014*.

[13] BECK, J., AND EICHMANN, D. Program and interface slicing for reverse engineering. In *Proceedings of the 15th International Conference on Software Engineering* (Los Alamitos, CA, USA, 1993), ICSE '93, IEEE Computer Society Press, pp. 509–518.

[14] BIEMAN, J. M., AND OTT, L. M. Measuring functional cohesion. *IEEE Transactions on Software Engineering 20* (1994), 644–657.

[15] BINKLEY, D. The application of program slicing to regression testing. In *Information and Software Technology Special Issue on Program Slicing* (1999), Elsevier, pp. 583–594.

[16] BINKLEY, D., AND HARMAN, M. A survey of empirical results on program slicing. *Advances in Computers 62* (2004).

[17] BISWAS, S. K. *Dynamic Slicing in Higher-order Programming Languages*. PhD thesis, University of Pennsylvania, Philadelphia, PA, USA, 1997.

[18] CANFORA, G., CIMITILE, A., LUCIA, A. D., AND LUCCA, G. A. D. Decomposing legacy programs: a first step towards migrating to client-server platforms. *Journal of Systems and Software 54*, 2 (2000), 99–110.

[19] CANFORA, G., DE LUCIA, A., DI LUCCA, G., AND FASOLINO, A. Slicing large programs to isolate reusable functions, 10 1994.

[20] CHENEY, C. J. A nonrecursive list compacting algorithm. *Commun. ACM 13*, 11 (Nov. 1970), 677–678.

[21] Chitil, O. Type inference builds a short cut to deforestation. In *ICFP* (1999).

[22] Cimitile, A., Lucia, A. D., and Munro, M. Identifying reusable functions using specification driven program slicing: a case study. In *Proceedings of the International Conference on Software Maintenance, ICSM 1995, Opio (Nice), France, October 17-20, 1995* (1995), pp. 124–133.

[23] Clinger, W. D. Proper tail recursion and space efficiency. *SIGPLAN Not. 33*, 5.

[24] Emami, M., Ghiya, R., and Hendren, L. J. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation* (New York, NY, USA, 1994), PLDI '94, ACM, pp. 242–256.

[25] Fairbairn, J., and Wray, S. *Functional Programming Languages and Computer Architecture: Portland, Oregon, USA, September 14–16, 1987 Proceedings*. Springer Berlin Heidelberg, 1987, ch. Tim: A simple, lazy abstract machine to execute supercombinators, pp. 34–45.

[26] Fenichel, R. R., and Yochelson, J. C. A lisp garbage-collector for virtual-memory computer systems. *Commun. ACM 12*, 11 (Nov. 1969), 611–612.

[27] Ferrante, J., Ottenstein, K. J., and Warren, J. D. *The program dependence graph and its use in optimization*. Springer Berlin Heidelberg, 1984, pp. 125–132.

[28] Gharat, P. M., Khedker, U. P., and Mycroft, A. Flow- and context-sensitive points-to analysis using generalized points-to graphs. In *Static Analysis - 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings* (2016), pp. 212–236.

[29] Gill, A., Launchbury, J., and Peyton-Jones, S. L. A short cut to deforestation. In *FPCA* (1993).

[30] Gupta, R., Jean, M., Harrold, M. J., and Soffa, M. L. An approach to regression testing using slicing. In *In Proceedings of the Conference on Software Maintenance* (1992), IEEE Computer Society Press, pp. 299–308.

[31] Hamilton, G. W. Compile-time garbage collection for lazy functional languages. In *IWMM* (1995).

[32] Harman, M., and Danicic, S. Using program slicing to simplify testing.

[33] Harrold, M. J., and Soffa, M. L. Interprocedual data flow testing. In *Proceedings of the ACM SIGSOFT '89 Third Symposium on Software Testing, Analysis, and Verification* (New York, NY, USA, 1989), TAV3, ACM, pp. 158–167.

[34] Hindley, R. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society 146* (1969), 29–60.

[35] Hirzel, M., Diwan, A., and Henkel, J. On the usefulness of type and liveness accuracy for garbage collection and leak detection. *TOPLAS* (2002).

[36] Hofmann, M., and Jost, S. Static prediction of heap space usage for first-order functional programs. In *POPL* (2003).

[37] Horwitz, S., Prins, J., and Reps, T. Integrating noninterfering versions of programs. *ACM Trans. Program. Lang. Syst. 11*, 3 (July 1989), 345–387.

[38] Horwitz, S., Reps, T., and Binkley, D. Interprocedural slicing using dependence graphs. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation* (1988), PLDI '88.

[39] Horwitz, S., Reps, T. W., and Binkley, D. Interprocedural slicing using dependence graphs. In *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI) 1988.*

[40] Hunt, S., and Sands, D. Binding time analysis: A new perspective. In *Proceedings of the 1991 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation* (New York, NY, USA, 1991), PEPM '91, ACM, pp. 154–165.

[41] Inoue, K., Seki, H., and Yagi, H. Analysis of functional programs to detect run-time garbage cells. *TOPLAS* (1988).

[42] Kanade, A., Khedker, U., and Sanyal, A. Heterogeneous fixed points with application to points-to analysis. In *Proceedings of the Third Asian Conference on Programming Languages and Systems* (Berlin, Heidelberg, 2005), APLAS'05, Springer-Verlag, pp. 298–314.

[43] Kang, B.-K., and Bieman, J. M. Using design abstractions to visualize, quantify, and restructure software. *J. Syst. Softw. 42*, 2 (Aug. 1998), 175–187.

[44] Karkare, A., Khedker, U., and Sanyal, A. Liveness of heap data for functional programs. In *Heap Analysis and Verification, HAV 2007* (2007).

[45] Karkare, A., Sanyal, A., and Khedker, U. Effectiveness of garbage collection in MIT/GNU Scheme.

[46] Kennedy, K. A global flow analysis algorithm. *International Journal of Computer Mathematics 3*, 1-4 (1972), 5–15.

[47] Kennedy, K. W. Node listings applied to data flow analysis. In *Proceedings of the 2Nd ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (New York, NY, USA, 1975), POPL '75, ACM, pp. 10–21.

[48] Khedker, U. P., Mycroft, A., and Rawat, P. S. Liveness-based pointer analysis. In *Static Analysis - 19th International Symposium, SAS 2012, Deauville, France, September 11-13, 2012. Proceedings* (2012), pp. 265–282.

[49] Khedker, U. P., Sanyal, A., and Karkare, A. Heap reference analysis using access graphs. *TOPLAS* (2007).

[50] Krinke, J., and Snelting, G. Validation of measurement software as an application of slicing and constraint solving1a preliminary version of parts of this article appeared in the proceedings of the third static analysis symposium [16].1. *Information and Software Technology 40*, 11 (1998), 661 – 675.

[51] Lakhotia, A., and christophe Deprez, J. Restructuring programs by tucking statements into functions, 1999.

[52] Landi, W., and Ryder, B. G. A safe approximate algorithm for interprocedural aliasing. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming*

Language Design and Implementation (New York, NY, USA, 1992), PLDI '92, ACM, pp. 235–248.

[53] Landi, W., Ryder, B. G., and Zhang, S. Interprocedural modification side effect analysis with pointer aliasing. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation* (New York, NY, USA, 1993), PLDI '93, ACM, pp. 56–67.

[54] Lanubile, F., and Visaggio, G. Extracting reusable functions by flow graph-based program slicing. *IEEE Trans. Softw. Eng. 23*, 4 (Apr. 1997), 246–259.

[55] Lee, O., Yang, H., and Yi, K. Inserting safe memory reuse commands into ML-like programs. In *SAS* (2003).

[56] Lee, O., Yang, H., and Yi, K. Static insertion of safe and effective memory reuse commands into ML-like programs. *Science of Computer Programming* (2005).

[57] Liu, Y. A., and Stoller, S. D. Eliminating dead code on recursive data. *Sci. Comput. Program. 47* (2003).

[58] Lyle, J. R., and Weiser, M. Automatic Program Bug Location by Program Slicing. In 2^{nd} *International Conference on Computers and Applications* (Peking, 1987), IEEE Computer Society Press, Los Alamitos, California, USA, pp. 877–882.

[59] Milner, R. A theory of type polymorphism in programming. *Journal of Computer and System Sciences 17*, 3 (1978), 348 – 375.

[60] Mitchell, N., and Runciman, C. Losing functions without gaining data: Another look at defunctionalisation. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell* (2009).

[61] Mogensen, T. A. Binding time analysis for polymorphically typed higher order languages. In *Proceedings of the International Joint Conference on Theory and Practice of Software Development, Volume 2: Advanced Seminar on Foundations of Innovative Software Development II and Colloquium on Current Issues in Programming Languages* (Berlin, Heidelberg, 1989), TAPSOFT '89, Springer-Verlag, pp. 298–312.

[62] Mohnen, M. Efficient compile-time garbage collection for arbitrary data structures. In *PLILPS* (1995).

[63] Mohri, M., and Nederhof, M.-J. Regular approximation of context-free grammars through transformation. In *Robustness in Language and Speech Technology*. Kluwer Academic Publishers, 2000.

[64] Nederhof, M.-J., and Satta, G. The language intersection problem for non-recursive context-free grammars. *Inf. Comput. 192*, 2 (Aug. 2004), 172–184.

[65] Nielson, H. R., and Nielson, F. Automatic binding time analysis for a typed Λ-calculus. *Sci. Comput. Program. 10*, 2 (Apr. 1988), 139–176.

[66] NoFib. Haskell Benchmark Suite. `http://git.haskell.org/nofib.git`, Feb 2017. (Last accessed).

[67] Ochoa, C., Silva, J., and Vidal, G. Dynamic slicing of lazy functional programs based on redex trails. *Higher Order Symbol. Comput. 21* (2008).

[68] O'Neill, M. E., and Burton, F. W. Smarter garbage collection with simplifiers. In *MSPC* (2006).

[69] Ott, L. Using slice profiles and metrics during software maintenance. In *In Proceedings of the 10th Annual Software Reliability Symposium* (1992), pp. 16–23.

[70] Ott, L. M., and Bieman, J. M. Program slices as an abstraction for cohesion measurement. *Information and Software Technology 40*, 11 (1998), 691 – 699.

[71] Ott, L. M., and Thuss, J. J. The relationship between slices and module cohesion. In *Proceedings of the 11th International Conference on Software Engineering* (New York, NY, USA, 1989), ICSE '89, ACM, pp. 198–204.

[72] Ottenstein, K. J., and Ottenstein, L. M. The program dependence graph in a software development environment. *ACM SIGPLAN Notices 19* (1984).

[73] Perera, R., Acar, U. A., Cheney, J., and Levy, P. B. Functional programs that explain their work. In *ACM SIGPLAN International Conference on Functional Programming, ICFP 2012*.

[74] Peyton-Jones, S. L. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.

[75] Peyton-Jones, S. L., and Metayer, D. L. Compile-time garbage collection by sharing analysis. In *FPCA* (1989).

[76] Pierce, B. C. *Types and Programming Languages*, 1st ed. The MIT Press, 2002.

[77] Reps, T. Program analysis via graph reachability. In *Proceedings of the 1997 International Symposium on Logic Programming* (Cambridge, MA, USA, 1997), ILPS '97, MIT Press, pp. 5–19.

[78] Reps, T. Undecidability of context-sensitive data-dependence analysis. *ACM Trans. Program. Lang. Syst. 22*, 1 (Jan. 2000), 162–186.

[79] Reps, T. W., and Turnidge, T. Program specialization via program slicing. In *Partial Evaluation, International Seminar, Dagstuhl Castle, Germany* (1996).

[80] Rodrigues, N. F., and Barbosa, L. S. Component identification through program slicing. *Electronic Notes in Theoretical Computer Science 160* (2006).

[81] Rodrigues, N. F., and Barbosa, L. S. Program slicing by calculation. *Journal of Universal Computer Science* (2006).

[82] Röjemo, N., and Runciman, C. Lag, drag, void and use—heap profiling and space-efficient compilation revisited. In *ICFP* (1996).

[83] Sabry, A., and Felleisen, M. Reasoning about programs in continuation-passing style. *SIGPLAN Lisp Pointers* (1992).

[84] Sagiv, M., Reps, T., and Wilhelm, R. Parametric shape analysis via 3-valued logic. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (1999), POPL '99, ACM, pp. 105–118.

[85] Sagiv, M., Reps, T., and Wilhelm, R. Shape analysis and applications. In *Compiler Design Handbook: Optimizations and Machine Code Generation*, Y. N. Srikant and P. Shankar, Eds. CRC Press, Inc, 2002.

[86] Saswat Padhi. BTP Report. `https://www.cse.iitb.ac.in/~saswatpadhi10/btp/report.pdf`, March 2014. (Last accessed).

[87] Shaham, R., Kolodner, E. K., and Sagiv, M. Heap profiling for space-efficient java. In *PLDI* (2001).

[88] Shaham, R., Kolodner, E. K., and Sagiv, M. Estimating the impact of heap liveness information on space consumption in Java. In *ISMM* (2002).

[89] Shaham, R., Yahav, E., Kolodner, E. K., and Sagiv, S. Establishing local temporal heap safety properties with applications to compile-time memory management. In *SAS* (2003).

[90] Shao, Z., and Appel, A. W. Efficient and safe-for-space closure conversion. *TOPLAS 22* (2000).

[91] Shivers, O. Control flow analysis in scheme. In *PLDI '88* (1988).

[92] Silva, J. A vocabulary of program slicing-based techniques. *ACM Comput. Surv.* (2012).

[93] Silva, J., Tamarit, S., and Tomás, C. System dependence graphs in sequential erlang. In *Proceedings of the 15th International Conference on Fundamental Approaches to Software Engineering* (2012), FASE'12.

[94] Soo Kim, H., Rae Kwon, Y., and Chung, I. Restructuring programs through program slicing. 349–368.

[95] Steensgaard, B. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 1996), POPL '96, ACM, pp. 32–41.

[96] Thomas, S. Garbage collection in shared-environment closure reducers: Space-efficient depth first copying using a tailored approach. *Information Processing Letters 56*, 1 (1995), 1 – 7.

[97] Tip, F. A survey of program slicing techniques. *Journal of Programming Languages 3* (1995).

[98] Tofte, M., and Birkedal, L. A region inference algorithm. *TOPLAS* (1998).

[99] Wadler, P. Deforestation: transforming programs to eliminate trees. In *ESOP* (1988).

[100] Wadler, P. Strictness analysis aids time analysis. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 1988), POPL '88, ACM, pp. 119–132.

[101] Wadler, P., and Hughes, J. Projections for strictness analysis. In *Proceedings of the 1987 Conference on Functional Programming Languages and Computer Architecture* (Portland, Oregon, September 1987).

[102] Weiser, M. Programmers use slices when debugging. *Commun. ACM 25*, 7 (July 1982), 446–452.

[103] Weiser, M. Program slicing. *IEEE Trans. Software Eng. 10* (1984).

[104] Yannakakis, M. Graph-theoretic methods in database theory. In *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (New York, NY, USA, 1990), PODS '90, ACM, pp. 230–242.